AFEM

C. Carstensen, J. Gedicke, L. Kern, J. Neumann,
H. Rabus, M. Rozova

February 15, 2010 – rev. 577

Contents

Acknowledgements

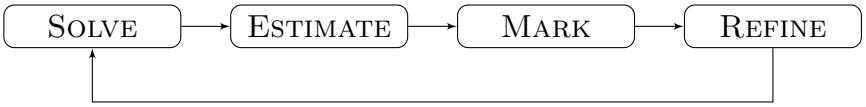
This work is based on various software developed in Prof. Carstensen's group within the past years at Humboldt-Universität zu Berlin and Technische Universität Wien with the help of C. Bahriawati, A. Byfut, J. Gedicke, D. Günther, J. Reininghaus, J. Thiele and S. Wiedemann.

1 Documentation

1.1 Introduction

The AFEM software package provides a flexible and easy-to-use toolbox for the implementation of adaptive finite element methods (AFEM). Unlike existing black-box solutions, the implementation of this package is transparent and easy to comprehend while non-object oriented. In order to retain simplicity, all examples are restricted to the Poisson model problem and the Stokes' equations, both introduced in Section 1.2.1. However, the programs can be easily modified for more complicated problems. The general structure of adaptive finite element methods is the AFEM cycle.

Figure 1.1.1: The AFEM cycle



This package includes functions realising each step of the AFEM cycle for the Poisson model problem using the conforming Courant (P_1), the non-conforming Crouzeix-Raviart (CR) and the mixed Raviart-Thomas (RT_0) elements and for the Stokes' equations using the non-conforming Crouzeix-Raviart element. Therefore, all necessary data structures as well as general marking and refining routines are provided. Further problems and finite elements, as well as other marking and refinement strategies, may be added by the user.

This documentation is organised as follows. Section 1.2 is devoted to the Poisson model problem, the Stokes' equations and the representation of the problem data in the software. In Section 1.3, the concept of regular triangulations is introduced and the representation and usage of the geometrical data structures and their applications are covered. The implementations of the AFEM cycle for the Poisson model problem with the P_1, CR and RT_0 finite elements and for the Stokes' equations with the CR finite element are described in detail in the subsequent sections. Section 1.4 describes the implementation of the step SOLVE. Here the approximative solution, which depends on the specific finite element and the model problem, as well as the local and global stiffness matrices and the right-hand side of the linear system, are computed. In step ESTIMATE, the a posteriori error estimator for a given discrete solution of the model problem for the corresponding triangulation is computed (cf. Section 1.5). This estimate is used by MARK to assign sides or elements for refinement in REFINE on the basis of different criteria, e.g., the bulk or maximum criterion (cf. Section 1.6). In step REFINE, the mesh

is refined after a closure algorithm ensures the preservation of the mesh quality. The implemented refinement strategies such as *Red-Green-Blue* or *Newest-Vertex Bisection* are introduced in Section 1.7. The result is a finer mesh that can be passed on to SOLVE for the next iteration of the AFEM cycle. Section 1.8 introduces some mathematical notations and theoretical background of jumps and averaging, numerical quadrature, the L^2 Norm and oscillations. The utility functions provide supplemental structural data as well as visualisation routines (Section 1.9). These include enumeration functions, which provide all necessary data structures to handle the mesh. Although there is no canonical enumeration, it remains consistent throughout the framework as long as the mesh structure is not altered. The plotting capabilities include the graphical analysis of the convergence rate, the mesh geometry and the discrete solutions of P_1, CR and RT_0 finite elements.

Chapter 2 contains example implementations of the AFEM cycle for P_1, CR and RT_0 finite elements whereas Chapter 3 contains the complete source code as a reference.

1.2 Model Problems

This section introduces the implemented model problems and their corresponding data structures.

1.2.1 Poisson Problem

The Poisson problem is given as follows. Let $\Omega \subset \mathbb{R}^2$ be a domain with piecewise linear boundary $\partial\Omega$, let $\Gamma_D \subseteq \partial\Omega$ be the closed Dirichlet boundary and $\Gamma_N := \partial\Omega \setminus \Gamma_D$ the Neumann boundary. Moreover, let $f \in L^2(\Omega)$, $u_D \in H^1(\Omega)$ and $g \in L^2(\Gamma_N)$. The Poisson model problem reads: Find $u \in C^2(\Omega)$ satisfying

$$-\Delta u = f, \quad u = u_D \text{ on } \Gamma_D \quad \text{and} \quad \frac{\partial u}{\partial \nu} = g \text{ on } \Gamma_N, \tag{1.2.1}$$

where ν is the outer unit normal vector on Γ_N. Thus, the weak formulation of (1.2.1) reads: Find $u \in H^1_D(\Omega)$ such that for all $v \in H^1_0(\Omega)$

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\Gamma_N} g v \, ds - \int_\Omega \nabla u_D \cdot \nabla v \, dx. \tag{1.2.2}$$

1.2.2 Stokes' Equations

Stokes' equations occur as a simplification of the Navier-Stokes equation and model the stationary, viscous, incompressible flow of a fluid through a domain $\Omega \subset \mathbb{R}^2$ with the outer unit normal vector ν. $\partial\Omega$ is assumed to be piecewise linear and to consist of a Dirichlet boundary part Γ_D and a Neumann boundary part $\Gamma_N = \partial\Omega \setminus \Gamma_D$. At each point $x \in \Omega$ the velocity $u : \Omega \to \mathbb{R}^2$ and the pressure $p : \Omega \to \mathbb{R}$ are modelled. The

values of $f : \Omega \to \mathbb{R}^2$ represent an outer body force, such as gravity. The Stokes problem reads: Find $u \in C^2(\Omega; \mathbb{R}^2)$ and $p \in C^1(\Omega)$ such that

$$\begin{aligned} -\Delta u + \nabla p &= f && \text{in } \Omega, \\ \operatorname{div} u &= 0 && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ (\nabla u - pI_2) \cdot \nu &= g && \text{on } \Gamma_N, \end{aligned} \tag{1.2.3}$$

where I_2 denotes the 2×2-dimensional identity matrix and $\sigma := \nabla u - pI_2$ the so-called pseudostress. Then the Gauss divergence theorem yields a necessary condition for the existence of some solution which reads

$$\int_{\partial\Omega} u_D \cdot \nu \,\mathrm{ds} = \int_{\partial\Omega} u \cdot \nu \,\mathrm{ds} = \int_\Omega \operatorname{div} u \,\mathrm{dx} = 0.$$

Multiplication of the first equation in (1.2.3) with a test function $v \in H^1(\Omega; \mathbb{R}^2)$ and integration by parts lead to the following weak formulation: Find $(u, p) \in H^1(\Omega; \mathbb{R}^2) \times L^2(\Omega)$ such that $u_{|\Gamma_D} = u_D$ (in the sense of traces) and

$$\begin{aligned} \int_\Omega \nabla u : \nabla v \,\mathrm{dx} - \int_\Omega p \operatorname{div} v \,\mathrm{dx} &= \int_\Omega f \cdot v \,\mathrm{dx} + \int_{\Gamma_N} g \cdot v \,\mathrm{ds}, \\ \int_\Omega q \operatorname{div} u \,\mathrm{dx} &= 0, \end{aligned} \tag{1.2.4}$$

for all $(v, q) \in H^1_D(\Omega; \mathbb{R}^2) \times L^2_0(\Omega)$. If $\int_{\Gamma_N} 1 \,\mathrm{dx} = 0$ then p is only defined up to a constant and is sought for in $L^2_0(\Omega) = \{q \in L^2(\Omega) \mid \int_\Omega q \,\mathrm{dx} = 0\}$.

From a mathematical point of view the pressure p turns out to be the Lagrangian multiplier for the side condition on the divergence of the velocity u. [7, section 6]

1.2.3 Problem Input Data

In order to describe the model problems in the software, the input data has to be supplied by the user. That is the function f as well as the values for the Dirichlet and Neumann boundary. In the case of the Stokes problem, `f`, `uDb` and `g` have to return values in $\mathbb{R}^2$.

Load – `f(x)`

f denotes the function on the right-hand side of the model problem (1.2.1). The input is a matrix `x` of points at which f is to be evaluated. It has one row for each point containing its two coordinates. The output is a vector containing the values of f at the given points.

In `afemP1PoissonShort.m` $f \equiv 1$ is implemented as an example for problem (1.2.1).

```
28 function val = f(x)
       val = ones(size(x,1),1);
30 end
```

u on the Dirichlet Boundary – `u4Db(x)`

`u4Db` represents the Dirichlet boundary condition u_D. It returns the (fixed) values of the solution u for points on the Dirichlet boundary. The input is a matrix `x` of points in Γ_D at which u_D is to be evaluated. It has one row for each point containing the two coordinates. The output is a vector containing the values of u_D at the given points.

As an example the implementation of $u_D \equiv 0$ can be found in `afemP1PoissonShort.m`.

```
32 function val = u4Db(x)
       val = zeros(size(x,1),1);
   end
```

Values for the Neumann Boundary – `g(x)`

To include Neumann boundary conditions, the flux of u in the direction of the outer unit normals of Ω has to be evaluated. Therefore, the input of `g` is a matrix `x` of points in Γ_N. The output is a vector containing the values of `g` at the given points in their corresponding outer unit normal directions.

A possible implementation for $g \equiv 1$ is given in `afemP1PoissonShort.m`.

```
36 function val = g(x)
       val = ones(size(x,1),1);
   end
```

1.3 Data Structures

This section introduces regular triangulations as well as their corresponding data structures.

1.3.1 Triangulation of the Domain Ω

Adaptive finite element methods require the domain $\Omega \subset \mathbb{R}^2$ to be decomposed into a finite number of triangles or quadrilaterals. In AFEM, all triangulations are assumed to be regular as defined below.

Definition 1.3.1. A *triangulation* $\mathcal{T}_\ell$ of a domain Ω is a finite set of triangles, such that

$$\bigcup \mathcal{T}_\ell = \overline{\Omega},$$

where every $T \in \mathcal{T}_\ell$ is a triangle $T = \operatorname{conv}\{z_1, z_2, z_3\}$ for $z_1, z_2, z_3 \in \Omega$ not collinear. The triangles are often referred to as *elements*. The vertices of T are called *nodes*. They are collected in the set $\mathcal{N}(T) = \{z_1, z_2, z_3\}$ and the set of nodes of the entire triangulation is denoted by

$$\mathcal{N}_\ell = \bigcup_{T \in \mathcal{T}_\ell} \mathcal{N}(T).$$

In addition, $\mathcal{K}_\ell$ is the set of inner nodes (nodes which are not in $\partial\Omega$). The edges of T are called *sides* and the sets of sides of T and $\mathcal{T}_\ell$ are given by

$$\mathcal{S}(T) = \{S_1, S_2, S_3\} \quad \text{and} \quad \mathcal{S}_\ell = \bigcup_{T\in\mathcal{T}_\ell} \mathcal{S}(T),$$

where $S_j = \operatorname{conv}\{z_j, z_{j+1}\}$ (j is to be understood modulo 3). The length of a side S is denoted by h_S and the sets of boundary sides are denoted by $\mathcal{S}_{\ell,D}$ and $\mathcal{S}_{\ell,N}$ for Dirichlet and Neumann boundary, respectively. The set $\mathcal{S}_{\ell,\Omega}$ contains all interior sides of $\mathcal{T}_\ell$.

Definition 1.3.2. A triangulation $\mathcal{T}_\ell$ of a domain Ω is called *regular* if (and only if) for two distinct triangles $T_1, T_2 \in \mathcal{T}_\ell$ the intersection $T_1 \cap T_2$ is either empty or a single common node or a common side of T_1 and T_2, i.e.,

$$T_1 \cap T_2 \in \mathcal{N}_\ell \cup (\mathcal{S}(T_1) \cap \mathcal{S}(T_2)).$$

Definition 1.3.3. For a node $z \in \mathcal{N}_\ell$ of a triangulation $\mathcal{T}_\ell$ the *node patch* ω_z of z is given by

$$\omega_z := \operatorname{int} \bigcup_{\substack{T\in\mathcal{T}_\ell \\ z\in\mathcal{N}(T)}} T.$$

Similarly, the *patch of a side* ω_S for $S \in \mathcal{S}_\ell$ is defined as

$$\omega_S := \operatorname{int} \bigcup_{\substack{T\in\mathcal{T}_\ell \\ S\in\mathcal{S}(T)}} T.$$

Extended node patches Ω_z for interior nodes $z \in \mathcal{K}_\ell$, realise a finite overlay of Ω with patches containing at least one inner node. Let $\mathcal{T}_\ell$ be a triangulation where each boundary node has a neighbouring inner node. Let $\zeta : \mathcal{N}_\ell \to \mathcal{K}_\ell$ be a mapping where inner nodes are fixed points, while each boundary node is mapped to one of the neighbouring inner nodes. The extended node patch of an interior node $z \in \mathcal{K}_\ell$ is defined as

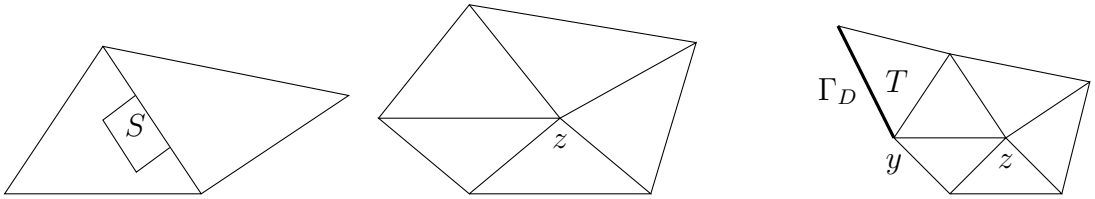
$$\Omega_z := \bigcup_{\substack{z=\zeta(y) \\ y\in\mathcal{N}_\ell}} \omega_y.$$

Figure 1.3.1 shows examples for a side patch, a node patch and an extended node patch.

1.3.2 Geometrical Data

The following data structures store information about the coordinates, elements and boundary sides of a triangulation. An example triangulation of an L-shaped domain is given in Figure 1.3.3.

Figure 1.3.1: Illustration of side patches, node patches and extended node patches.



(a) Side patch of $S \in \mathcal{S}_\ell$ (b) Node patch of $z \in \mathcal{N}_\ell$ (c) Example of an extended node patch Ω_z of z with $\zeta(y) = z$ and $\omega_z = \Omega_z \setminus T$

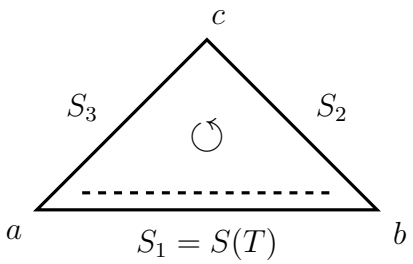
Coordinates for Nodes – `c4n`

`c4n` is an $|\mathcal{N}_\ell| \times 2$ matrix, which stores the coordinates of all nodes of a triangulation. For each node there is one row in `c4n` containing its x- and y-coordinate. Note, that the order of the rows is not relevant. However it implies a numbering of the nodes given through the respective row numbers.

Nodes for Elements – `n4e`

`n4e` is a $|\mathcal{T}_\ell| \times 3$ matrix. Each row contains the node numbers for a triangle $T = \operatorname{conv}\{a, b, c\} \in \mathcal{T}_\ell$. The nodes are ordered counterclockwise w.r.t. the corresponding element. For technical reasons, the first two nodes are the endpoints of the reference side $S(T)$, cf. Figure 1.3.2. In the initial triangulation $\mathcal{T}_0$ this is preferred to be the longest side of T. This has to be imposed by the user.

Figure 1.3.2: Counterclockwise enumeration of nodes and sides of a triangle $T \in \mathcal{T}_\ell$.



Nodes for Dirichlet Boundary – `n4sDb`

`n4sDb` is a $|\mathcal{S}_{\ell,D}| \times 2$ matrix storing the Dirichlet boundary sides of $\mathcal{T}_\ell$. For each such side $S \in \mathcal{S}_{\ell,D}$, there is one row in `n4sDb` containing the node numbers of S ordered counterclockwise w.r.t. the element which the side is a part of. For the example considered in

Figure 1.3.3, `n4sDb` may look as follows

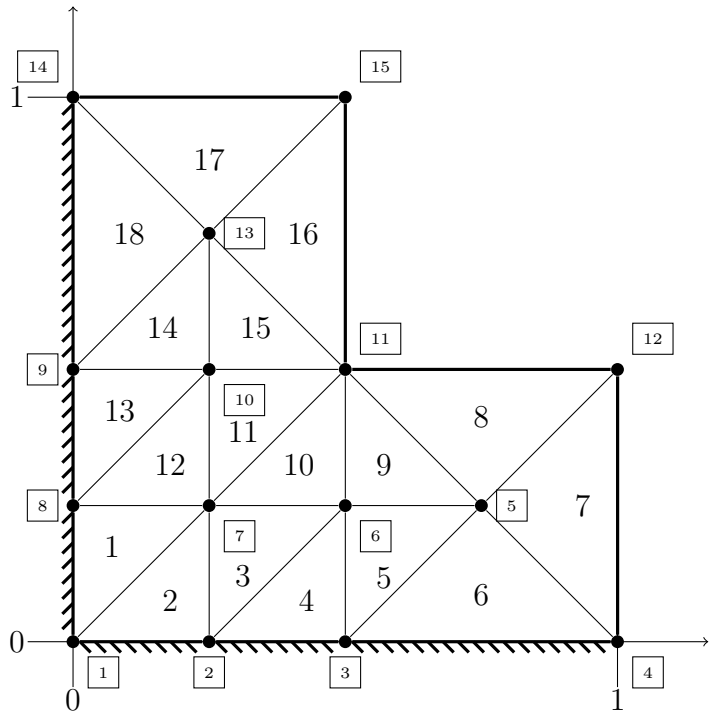
$$\texttt{n4sDb} = \begin{pmatrix} 1 & 2 & 3 & 14 & 9 & 8 \\ 2 & 3 & 4 & 9 & 8 & 1 \end{pmatrix}^T.$$

Nodes for Neumann Boundary – `n4sNb`

`n4sNb` is an $|\mathcal{S}_{\ell,N}| \times 2$ matrix storing the Neumann boundary sides of $\mathcal{T}_\ell$. For each side in $\mathcal{S}_{\ell,N}$, there is one row in `n4sNb` containing the node numbers of that side ordered counterclockwise with respect to the element which the side is a part of. For the example considered in Figure 1.3.3, the remaining boundary sides are stored in `n4sNb`

$$\texttt{n4sNb} = \begin{pmatrix} 4 & 12 & 11 & 15 \\ 12 & 11 & 15 & 14 \end{pmatrix}^T.$$

Figure 1.3.3: One possible triangulation of an L-shaped domain. Dirichlet boundary sides have stripes, Neumann boundary sides are thicker.



(a) $\mathcal{T}_\ell$

0	0
0.25	0
0.5	0
1	0
0.75	0.25
0.5	0.25
0.25	0.25
0	0.25
0	0.5
0.25	0.5
0.5	0.5
1	0.5
0.25	0.75
0	1
0.5	1

(b) `c4n`

1	7	8
7	1	2
2	6	7
6	2	3
3	5	6
3	4	5
4	12	5
12	11	5
5	11	6
11	7	6
7	11	10
10	8	7
8	10	9
13	9	10
11	13	10
11	15	13
15	14	13
14	9	13

(c) `n4e`

1.4 SOLVE

In the first step of the AFEM cycle the discrete solution of a given problem on the current mesh is computed. The following functions implement this step for conforming

(e.g. `solveP1Poisson`) and non-conforming (e.g. `solveCRPoisson`) P_1 finite elements as well as mixed RT_0 finite elements (e.g. `solveRT0Poisson`). Throughout this section, $P_k(\mathcal{T}_\ell)$ denotes the set of piecewise polynomial functions of maximal degree $k \geq 0$ over $\mathcal{T}_\ell$. That is

$$P_k(\mathcal{T}_\ell) = \left\{ v \in L^2(\Omega) \mid \forall T \in \mathcal{T}_\ell : v|_T \text{ is a polynomial of maximal degree } k \right\}. \quad (1.4.1)$$

1.4.1 P_1 Solver for the Poisson Problem

`[x,nrDof,A,b] = solveP1Poisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)`

Let $V_C(\mathcal{T}_\ell) := P_1(\mathcal{T}_\ell) \cap C(\overline{\Omega})$ be the Courant finite element space spanned by a basis of piecewise linear functions φ_k, $1 \leq k \leq |\mathcal{N}_\ell|$ which equal one at node $z_k \in \mathcal{N}_\ell$ and zero at the other nodes. The discrete version of the Poisson problem (1.2.2) with f, g, u_D for some $u_{\ell,D} \in V_C(\mathcal{T}_\ell)$ satisfying $u_{\ell,D} = u_D$ on $\mathcal{N}_\ell \cap \Gamma_D$ and the homogeneous solution $u_{\ell,0} \in V_C^0(\mathcal{T}_\ell) := \{v \in V_C(\mathcal{T}_\ell) : v_{\big|\Gamma_D} \equiv 0\}$ reads: Find $u_\ell = u_{\ell,D} + u_{\ell,0} \in V_C(\mathcal{T}_\ell)$ such that for all $v_\ell \in V_C^0(\mathcal{T}_\ell)$

$$\int_\Omega \nabla u_{\ell,0} \cdot \nabla v_\ell \,\mathrm{dx} = \int_\Omega f v_\ell \,\mathrm{dx} + \int_{\Gamma_N} g v_\ell \,\mathrm{ds} - \int_\Omega \nabla u_{\ell,D} \cdot \nabla v_\ell \,\mathrm{dx}. \quad (1.4.2)$$

Non-zero Dirichlet conditions are incorporated by a decomposition of u_ℓ in $u_{\ell,0} + u_{\ell,D}$. As the φ_k form a basis of $V_C(\mathcal{T}_\ell)$, the discrete problem (1.4.2) can be rewritten as

$$\int_\Omega \nabla u_{\ell,0} \cdot \nabla \varphi_k \,\mathrm{dx} = \int_\Omega f \varphi_k \,\mathrm{dx} + \int_{\Gamma_N} g \varphi_k \,\mathrm{ds} - \int_\Omega \nabla u_{\ell,D} \cdot \nabla \varphi_k \,\mathrm{dx}, \quad (1.4.3)$$

for $1 \leq k \leq |\mathcal{N}_\ell|$. With coefficient vectors $(\hat{x}_1, \ldots, \hat{x}_{|\mathcal{N}_\ell|})^T$, $(\tilde{x}_1, \ldots, \tilde{x}_{|\mathcal{N}_\ell|})^T \in \mathbb{R}^{|\mathcal{N}_\ell|}$ for nodes $z_1, \ldots, z_{|\mathcal{N}_\ell|}$ such that

$$u_{\ell,0} = \sum_{z_k \in \mathcal{N}_\ell} \hat{x}_k \varphi_k \quad \text{and} \quad u_{\ell,D} = \sum_{z_k \in \mathcal{N}_\ell} \tilde{x}_k \varphi_k,$$

the discrete system (1.4.3) is equivalent to

$$\sum_{z_j \in \mathcal{N}_\ell} \hat{x}_j \int_\Omega \nabla \varphi_j \cdot \nabla \varphi_k \,\mathrm{dx} = \int_\Omega f \varphi_k \,\mathrm{dx} + \int_{\Gamma_N} g \varphi_k \,\mathrm{ds} - \sum_{z_j \in \mathcal{N}_\ell} \tilde{x}_j \int_\Omega \nabla \varphi_j \cdot \nabla \varphi_k \,\mathrm{dx} \quad (1.4.4)$$

and thus

$$\begin{aligned} \sum_{z_j \in \mathcal{K}_\ell} (\hat{x}_j + \tilde{x}_j) \int_\Omega \nabla \varphi_j \cdot \nabla \varphi_k \,\mathrm{dx} \; + \sum_{z_j \in \mathcal{N}_\ell \setminus \mathcal{K}_\ell} \hat{x}_j \int_\Omega \nabla \varphi_j \cdot \nabla \varphi_k \,\mathrm{dx} \\ = \int_\Omega f \varphi_k \,\mathrm{dx} + \int_{\Gamma_N} g \varphi_k \,\mathrm{ds} - \sum_{z_j \in \mathcal{N}_\ell \setminus \mathcal{K}_\ell} \tilde{x}_j \int_\Omega \nabla \varphi_j \cdot \nabla \varphi_k \,\mathrm{dx} \end{aligned} \quad (1.4.5)$$

for $1 \leq k \leq |\mathcal{N}_\ell|$. Since $\hat{x}_j = 0$ for nodes $z_j \in \mathcal{N}_\ell \backslash \mathcal{K}_\ell$ (i.e. nodes on the border), (1.4.5) can be rewritten as a linear system of equations $Ax = b$, with $A \in \mathbb{R}^{|\mathcal{N}_\ell| \times |\mathcal{N}_\ell|}$, $b \in \mathbb{R}^{|\mathcal{N}_\ell|}$ and $x = \hat{x} + \tilde{x}$ such that

$$A_{jk} := \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_k \,\mathrm{dx}, \tag{1.4.6}$$

$$b_k := \int_\Omega f\varphi_k \,\mathrm{dx} + \int_{\Gamma_N} g\varphi_k \,\mathrm{ds} - \sum_{z_j \in \mathcal{N}_\ell \backslash \mathcal{K}_\ell} \tilde{x}_j \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_k \,\mathrm{dx}. \tag{1.4.7}$$

Corresponding to (1.4.6) and (1.4.7), the assembling of the stiffness matrix A and the computation of the right-hand side b as well as the computation of the coefficient vector x for the solution $u_\ell \in V_C(\mathcal{T}_\ell)$ is implemented in `solveP1Poisson.m`. The input for this function is a mesh given by `n4e`, `c4n`, `n4sDb` and `n4sNb`, as well as the problem input data `f`, `g` and `u4Db`. The output is a coefficient vector `x` such that $u_\ell = \sum x_k\varphi_k$, the stiffness matrix A, the right-hand side b and the number of degrees of freedom `nrDof`.

The stiffness matrix is not assembled nodewise but elementwise. This results in linear instead of quadratic complexity.

For each element $T = \mathrm{conv}(\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}) \in \mathcal{T}_\ell$, the local stiffness matrix $A_{jk}(T)$ is generated by computing the gradients of its local basis functions $\hat{\varphi}_1, \hat{\varphi}_2, \hat{\varphi}_3$ as the solution of the linear system

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \begin{pmatrix} \nabla\hat{\varphi}_1 \\ \nabla\hat{\varphi}_2 \\ \nabla\hat{\varphi}_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{1.4.8}$$

For the right-hand side of the problem, $\int_T f\hat{\varphi}_k \,\mathrm{dx}$ for $1 \leq k \leq 3$ needs to be computed, and is approximated by the midpoint quadrature rule $\frac{|T|}{3} f(\mathrm{mid}(T))$, where $\mathrm{mid}(T)$ is the midpoint of T.

```
for elem = 1 : nrElems
    nodes = n4e(elem,:);
    coords = c4n(nodes,:);
    area = area4e(elem);
    mid = mid4e(elem,:);
    grads = [1,1,1;coords'] \ [0,0;eye(2)];
    b(nodes) = b(nodes)+(1/3) * area * f(mid)*[1;1;1];
    Alocal(:,:,elem) = area * grads * grads';
end
```

The values of (1.4.6) have been computed elementwise and need to be added to their respective elements in the global stiffness matrix which relate to the corresponding degree of freedom.

```
n4eT = n4e';
I = [n4eT;n4eT;n4eT];
J = [n4eT(:),n4eT(:),n4eT(:)]';
A = sparse(I(:),J(:),Alocal(:));
```

Afterwards, the Neumann and Dirichlet boundary conditions are incorporated by computing the corresponding integrals for the Neumann boundary from (1.4.7) and adding them to the right-hand side.

```
nrNbSides = size(n4sNb,1);
length4NbSides = computeLength4s(c4n,n4sNb);
mid4NbSides = computeMid4s(c4n,n4sNb);
for NbSide = 1 : nrNbSides
    nodes = n4sNb(NbSide,:);
    len = length4NbSides(NbSide);
    mid = mid4NbSides(NbSide);
    b(nodes) = b(nodes) + (1/2) * len * g(mid)*[1;1];
end
```

The components of the solution at the Dirichlet boundary are set to the given values of u_D on Γ_D.

```
x = zeros(nrNodes,1);
DbCoords = c4n(DbNodes,:);
x(DbNodes) = u4Db(DbCoords);
b = b - A * x;
```

Finally, the linear system is solved and the coefficient vector x of the discrete solution is returned.

```
x(dof) = A(dof,dof) \ b(dof);
```

1.4.2 Crouzeix-Raviart Solver for the Poisson Problem

`[x,nrDof,A,b] = solveCRPoisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)`

In some cases the conforming finite element spaces are too restrictive to approximate the solution for (1.2.2). Therefore one seeks an approximation of u in some finite dimensional non-conforming space, i.e., it is not included in $H^1_D(\Omega)$. For example continuity can be discarded.

The non-conforming Crouzeix-Raviart finite element space is defined as the space of piecewise linear functions on a triangulation $\mathcal{T}_\ell$ which are continuous across the midpoints of all sides (cf. [4]):

$$V_{CR}(\mathcal{T}_\ell) := P_1(\mathcal{T}_\ell) \cap C\left(\{\mathrm{mid}(S) \mid S \in \mathcal{S}_\ell\}\right).$$

This leads to the discretised weak form: Seek $u_\ell \in V_{CR}(\mathcal{T}_\ell)$ such that

$$\int_\Omega \nabla_\ell u_\ell \cdot \nabla_\ell v_\ell \,\mathrm{dx} = \int_\Omega f v_\ell \,\mathrm{dx} + \int_{\Gamma_N} g v_\ell \,\mathrm{ds}$$

for all $v_\ell \in V_{CR}(\mathcal{T}_\ell)$ where $u_\ell = u_{\ell,0} + u_{\ell,D}$ with $u_{\ell,0} \in V^0_{CR}(\mathcal{T}_\ell) := \{v \in V_{CR}(\mathcal{T}_\ell) \mid v(\mathrm{mid}(S)) = 0, S \in \mathcal{S}_{\ell,D}\}$, $u_{\ell,D} = u_D$ on $\Gamma_D \cap \mathrm{mid}(S_\ell)$ and with ∇_ℓ denoting the elementwise gradient operator. The discrete problem can be written as a linear system of equations $Ax = b$ using a basis $\{\varphi_k \mid 1 \le k \le |\mathcal{S}_\ell|\}$ of $V_{CR}(\mathcal{T}_\ell)$.

The function `solveCRPoisson` uses the problem data given by `f`, `g`, `u4Db` and the mesh data given by `c4n`, `n4e`, `n4sDb` and `n4sNb` as input and computes the vector `x` with the coefficients of the Crouzeix-Raviart basis functions for a discrete solution u_ℓ on the given mesh, where $u_\ell = \sum x_j \varphi_j$. Optionally the number of degrees of freedom `nrDof`, the stiffness matrix A and the right-hand side b are returned.

In order to get the stiffness matrix A, the local stiffness matrix A_T needs to be computed for each Element $T \in \mathcal{T}_\ell$:

$$A_{jk}(T) = \int_T \nabla \varphi_j \cdot \nabla \varphi_k \, \mathrm{dx},$$

where φ_k, $k \in \{1, 2, 3\}$, are the local Crouzeix-Raviart basis functions on T. As these functions are linear on each $T \in \mathcal{T}_\ell$, their gradients are piecewise constant and can be obtained in the following way. Let (x_1, y_1), (x_2, y_2), (x_3, y_3) be the vertices of T and let φ_k be of the form

$$\varphi_k(x, y) = a_k x + b_k y + c_k \text{ for some } a_k, b_k, c_k \in \mathbb{R}.$$

Then, $\varphi_k(x_j, y_j) = 1 - 2\delta_{kj}$ for $1 \leq j, k \leq 3$. In order to simplify computations, let $\tilde{\varphi}_k := \varphi_k - 1$, $\tilde{c}_k = c_k - 1$, thus

$$\tilde{\varphi}_k(x_j, y_j) = -2\delta_{kj}.$$

Note that $\nabla_\ell \varphi_k = \nabla_\ell \tilde{\varphi}_k = (a_k, b_k)^T$. This leads to the linear system of equations

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_1 & b_1 & \tilde{c}_1 \\ a_2 & b_2 & \tilde{c}_2 \\ a_3 & b_3 & \tilde{c}_3 \end{pmatrix} = \begin{pmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{pmatrix}.$$

As we are only interested in the a_j and b_j, the equations corresponding to $\tilde{c}_j$ are omitted. Thus, it remains to solve

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \nabla_\ell \varphi_1 \\ \nabla_\ell \varphi_2 \\ \nabla_\ell \varphi_3 \end{pmatrix} = \begin{pmatrix} -2 & 0 \\ 0 & -2 \\ 0 & 0 \end{pmatrix}.$$

In the following excerpt from `solveCRPoisson.m`, the corresponding Matlab code can be found. However, the numbering of the sides of the element is not the same as the numbering of the degrees of freedom. Thus, the gradients have to be reordered accordingly (see line 51).

```
46 nodes = n4e(elem,:);
   sides = s4e(elem,:);
   coords = c4n(nodes,:);
   area = area4e(elem);
50 grads = [coords';1 1 1]\[-2 0; 0 -2; 0 0];
   grads = grads([3 1 2],:);
   Alocal(:,:,elem) = area * grads * grads';
```

For the right-hand side of the problem $\int_T f\varphi_k \,\mathrm{dx}$ needs to be computed for $1 \leq k \leq 3$, which is approximated by the midpoint quadrature rule $\frac{|T|}{3} f(\mathrm{mid}(T))$ where $\mathrm{mid}(T)$ is the midpoint of T.

The corresponding Matlab code is realised in line 55 of `solveCRPoisson.m`.

```
53 mid = mid4e(elem,:);
   b(sides) = b(sides) + area*f(mid)*ones(3,1)/3;
```

The global stiffness matrix is assembled from the local ones by indexing over `s4e`. The following lines from `solveCRPoisson.m` show the corresponding Matlab code.

```
58 s4eT = s4e';
   I = [s4eT;s4eT;s4eT];
60 J = [s4eT(:),s4eT(:),s4eT(:)]';
   A = sparse(I(:),J(:),Alocal(:));
```

To include the Neumann boundary conditions, the integrals $\int_{S_N} g_\ell v_\ell \,\mathrm{ds}$ on the Neumann boundary sides $S_N \in \mathcal{S}_{\ell,N}$ are approximated by the midpoint quadrature rule $|S_N|\, g\left(\mathrm{mid}(S_N)\right)$. The solution on the Dirichlet boundary is set to the values given by u_D. After that, the system of linear equations is solved in line 72 of `solveCRPoisson.m`.

```
64 length4NbSides = computeLength4s(c4n,n4sNb);
65 mid4NbSides = computeMid4s(c4n,n4sNb);
   b(NbSides) = b(NbSides) + length4NbSides .* g(mid4NbSides);

   x = zeros(nrSides,1);
   x(DbSides) = u4Db(computeMid4s(c4n, n4sDb));
70 b = b - A * x;

   x(dof) = A(dof,dof) \ b(dof);
```

1.4.3 Raviart-Thomas Solver for the Poisson Problem

`[p,u,nrDof] = solveRT0Poisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)`

Another approach is the Raviart-Thomas finite element method. Sometimes only special properties of the solution are of interest, e.g., the flux. Therefore an additional flux variable $p := \nabla u$ is introduced. This results in the mixed formulation of (1.2.2) with f, g, u_D: Seek $(u, p) \in C^2(\Omega) \times C^1(\Omega)$ such that

$$\mathrm{div}\, p + f = 0 \text{ in } \Omega \qquad u = u_D \text{ on } \Gamma_D \qquad p \cdot n = g \text{ on } \Gamma_N.$$

Integration by parts leads to the weak dual mixed formulation: Seek $(u, p) \in L^2(\Omega) \times H(\mathrm{div}, \Omega)$ such that $\forall (v, q) \in L^2(\Omega) \times H(\mathrm{div}, \Omega)$ holds

$$\int_\Omega p \cdot q \,\mathrm{dx} + \int_\Omega u \,\mathrm{div}\, q \,\mathrm{dx} = \int_{\Gamma_D} u_D q \cdot \nu \,\mathrm{ds}$$
$$\int_\Omega v \,\mathrm{div}\, p \,\mathrm{dx} = -\int_\Omega f \cdot v \,\mathrm{dx}.$$

The Raviart-Thomas finite element space is defined as

$$\begin{aligned} V_{RT} &:= V_{RT}^{DC} \cap H(\mathrm{div},\Omega) \text{ with} \\ V_{RT}^{DC}(\mathcal{T}_\ell) &:= \{q \in L^2(\Omega;\mathbb{R}^2) : \\ &\qquad \forall T \in \mathcal{T}_\ell \; \exists a \in \mathbb{R}^2, c \in \mathbb{R} \; \forall x \in T : q_h(x)|_T = a + cx\} \text{ and} \\ H(\mathrm{div},\Omega) &:= \{v \in L^2(\Omega,\mathbb{R}^2) : \mathrm{div}\, v \in L^2(\Omega)\}. \end{aligned}$$

Note that the solution will be sought in the discontinious Raviart-Thomas space V_{RT}^{DC} whereas the normal jump of a Raviart-Thomas function $v \in V_{RT}$ vanishes across all sides in $\mathcal{S}_\ell$. The implementation of RT_0 MFEM outlined below is based on the Lagrange multiplier technique. For the complete theoretical background and detailed description of various implementations of RT_0 MFEM the reader is referred to [4].The discrete problem reads: Find $(p_\ell, u_\ell) \in V_{RT}^{DC}(\mathcal{T}_\ell) \times P_0(\mathcal{T}_\ell)$ with the Lagrange multipliers $(\alpha_\ell, \kappa_\ell) \in P_0(\mathcal{S}_\Omega) \times P_0(\mathcal{S}_N)$ such that

$$\begin{aligned} \int_\Omega p_\ell \cdot q_\ell \,\mathrm{dx} + \int_\Omega u_\ell \;\mathrm{div}\, q_\ell \,\mathrm{dx} - \sum_{S\in\mathcal{S}_\Omega} \int_S [q_\ell]_S \cdot \nu_S \alpha_\ell \,\mathrm{ds} - \sum_{S\in\mathcal{S}_N} \int_S q_\ell \cdot \nu_S \;\kappa_\ell \,\mathrm{ds} \\ = \int_{\Gamma_D} u_D \; q_\ell \cdot \nu \,\mathrm{ds}, \\ \int_\Omega v_\ell \;\mathrm{div}\, p_\ell \,\mathrm{dx} = \int_\Omega f \cdot v_\ell \,\mathrm{dx}, \\ -\sum_{S\in\mathcal{S}_\Omega} \int_S [p_\ell]_S \cdot \nu_S \beta_\ell \,\mathrm{ds} = 0, \\ \sum_{S\in\mathcal{S}_N} \int_S p_\ell \cdot \nu_S \; b_\ell \,\mathrm{ds} = -\int_{\Gamma_N} g \; b_\ell \,\mathrm{ds} \end{aligned} \tag{1.4.9}$$

for all $(q_\ell, v_\ell, \beta_\ell, b_\ell) \in V_{RT}^{DC}(\mathcal{T}_\ell) \times P_0(\mathcal{T}_\ell) \times P_0(\mathcal{S}_\Omega) \times P_0(\mathcal{S}_N)$, where $P_0(\mathcal{S}_\Omega)$ and $P_0(\mathcal{S}_N)$ are defined analogue to $P_0(\mathcal{T}_\ell)$ and $[.]_S$ denotes the jump as in Definition 1.8.1 on page 32.

Let x_T be the midpoint of the element T. Then $\psi_{T,1} \equiv (1,0)^T$, $\psi_{T,2} \equiv (0,1)^T$ and $\psi_{T,3} \equiv (x - x_T)$ for all $T \in \mathcal{T}_\ell$ are a basis of V_{RT}^{DC}. The discrete system can be rewritten in a system of linear equations

$$\begin{pmatrix} B & C & D & F \\ C^T & 0 & 0 & 0 \\ D^T & 0 & 0 & 0 \\ F^T & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} p_\ell \\ u_\ell \\ \alpha_\ell \\ \kappa_\ell \end{pmatrix} = \begin{pmatrix} b_D \\ b_f \\ 0 \\ b_g \end{pmatrix}, \tag{1.4.10}$$

with the matrices B, C, D and F as well as the vectors b_D, b_f and b_g as introduced in [4]. Exemplatory the assembling of the matrices B and C is explained in the sequel. The local stiffness matrices $B_T \in \mathbb{R}^{3\times3}$ and $C_T \in \mathbb{R}^{3\times1}$ for $T = \mathrm{conv}\{a,b,c\} \in \mathcal{T}_\ell$ are given

by

$$(B_T)_{jk} := \int_T \psi_j \cdot \psi_k \,\mathrm{dx} \quad \text{for } j,k = 1,2,3 \text{ and}$$
$$(C_T)_j := \int_T \operatorname{div} \psi_j \,\mathrm{dx} \quad \text{for } j = 1,2,3.$$

Let $s := |b-a|^2 + |c-b|^2 + |c-a|^2$ which results in

$$B_T = |T| \operatorname{diag}(1,1,s/36) \text{ and}$$
$$C_T = (0,0,2|T|)^T.$$

The submatrices $B \in \mathbb{R}^{3|\mathcal{T}_\ell| \times 3|\mathcal{T}_\ell|}$ and $C \in \mathbb{R}^{3|\mathcal{T}_\ell| \times |\mathcal{T}_\ell|}$ of the global stiffness matrix are assembled by

$$B = \begin{pmatrix} B_1 & 0 & \dots & 0 \\ 0 & B_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & B_{|\mathcal{T}_\ell|} \end{pmatrix} \text{ and } C = \begin{pmatrix} C_1 & 0 & \dots & 0 \\ 0 & C_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & C_{|\mathcal{T}_\ell|} \end{pmatrix}.$$

For further details and the complete definition of B, C, D, F, b_D, b_F, and b_g see [4]. The program `solveRT0Poisson.m` realises the problem with the right-hand side `f`, the boundary conditions `g` and `u4Db` as well as the mesh structure data `c4n`, `n4e`, `n4sDb` and `n4sNb`. The following code implements the global stiffness matrix as well as the boundary conditions. Details on the realisation of the boundary conditions can be found in [4, Sections 6.2.2 and 6.3.2]. Each submatrix of the stiffness matrix is initialised as a sparse matrix in the beginning.

```
43    B = sparse( 3*nrElems, 3*nrElems);
      C = sparse( 3*nrElems, nrElems);
```

The matrices B and C as well as b_f are computed.

```
51    for curElem = 1 : nrElems

          s = sum(length4s(s4e(curElem,:)).^2);

55        B( 3*curElem-[2,1,0],3*curElem-[2,1,0] ) = area4e(curElem) * diag([1,1,s/36]);
          C( 3*curElem-[2,1,0],curElem ) = [0;0;2*area4e(curElem)];

          b(3*nrElems+curElem) = -area4e(curElem) * f(mid4e(curElem,:));
      end
```

After the computation of D, F, b_D and b_g the global stiffness matrix is assembled.

```
93    O1 = sparse(size(C,2),size(C,2)+size(D,2)+size(F,2));
      O2 = sparse(size(D,2),size(C,2)+size(D,2)+size(F,2));
95    O3 = sparse(size(F,2),size(C,2)+size(D,2)+size(F,2));
      A = [B , C, D, F; ...
           C', O1 ; ...
```

```
        D', O2 ; ...
        F', O3 ];

100
    nrDof = nrISides + nrNbSides;
```

The solution `x` is computed by solving the linear system of equations (1.4.10) and splitted up into its components `p` and `u`. The latter two are returned.

```
104 x = A \ b;
105
    p = reshape(x(1:3*nrElems),3,nrElems)';
    u = x(3*nrElems+1:4*nrElems);
```

Optionally the number of degrees of freedom is returned with `[p,u,nrDof]`.

1.4.4 Crouzeix-Raviart-P_0 Solver for the Stokes' Equations

`[u,p,nrDof] = solveCRP0Stokes(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)`

Since $\operatorname{div} v_\ell = 0$ for $v_\ell \in V_C(\mathcal{T}_\ell;\mathbb{R}^2)$ holds only for $v_\ell \equiv 0$ on special triangulations, the conforming P_1 discretization is not applicable for the Stokes problem. Alternatively, one can choose the non-conforming Crouzeix-Raviart functions (see section 1.4.2) as discretization of the velocity field u and the piecewise constant functions for the pressure p. Then the discrete weak formulation of the Stokes problem reads: Seek $u_\ell \in V_{CR}(\mathcal{T}_\ell;\mathbb{R}^2)$ and $p_\ell \in P_0(\mathcal{T}_\ell) \cap L^2_0(\Omega)$ (i.e. $\int_\Omega p_\ell \,\mathrm{dx} = 0$), such that

$$\begin{aligned}\int_\Omega \nabla_\ell u_\ell : \nabla_\ell v_\ell \,\mathrm{dx} - \int_\Omega p_\ell \operatorname{div}_\ell v_\ell \,\mathrm{dx} &= \int_\Omega f \cdot v_\ell \,\mathrm{dx} + \int_{\partial\Omega} g \cdot v_\ell \,\mathrm{ds},\\ \int_\Omega q_\ell \operatorname{div}_\ell u_\ell \,\mathrm{dx} &= 0,\end{aligned}$$

for all $(v_\ell, q_\ell) \in V_{CR}(\mathcal{T}_\ell;\mathbb{R}^2) \times (P_0(\mathcal{T}_\ell) \cap L^2_0(\Omega))$, where $u_\ell = u_{\ell,0} + u_{\ell,D}$ with $u_{\ell,0} \in V^0_{CR}(\mathcal{T}_\ell)$ (as in section 1.4.2), $u_{\ell,D} = u_D$ on $\Gamma_D \cap \operatorname{mid}(S_\ell)$ and with ∇_ℓ and div_ℓ denoting the corresponding piecewise operators. The introduction of a Lagrangian multiplier $\lambda \in \mathbb{R}$ enforces the side condition on p. Finally the following weak formulation reads: Seek $u_\ell \in V_{CR}(\mathcal{T}_\ell;\mathbb{R}^2), p_\ell \in P_0(\mathcal{T}_\ell)$ and $\lambda \in \mathbb{R}$, such that

$$\begin{aligned}\int_\Omega \nabla_\ell u_\ell : \nabla_\ell v_\ell \,\mathrm{dx} - \int_\Omega p_\ell \operatorname{div}_\ell v_\ell \,\mathrm{dx} &= \int_\Omega f \cdot v_\ell \,\mathrm{dx} + \int_{\partial\Omega} g \cdot v_\ell \,\mathrm{ds},\\ \int_\Omega q_\ell \operatorname{div}_\ell u_\ell \,\mathrm{dx} + \lambda \int_\Omega q_\ell \,\mathrm{dx} &= 0,\\ \int_\Omega p_\ell \,\mathrm{dx} &= 0,\end{aligned} \tag{1.4.11}$$

for all $(v_\ell, q_\ell) \in V_{CR}(\mathcal{T}_\ell;\mathbb{R}^2) \times P_0(\mathcal{T}_\ell)$, where $u_\ell = u_{\ell,0} + u_{\ell,D}$ as above.

The problem results in a system of linear equations by using the edge-oriented basis functions $\{\varphi_k \mid 1 \le k \le |\mathcal{S}_\ell|\}$ of $V_{CR}(\mathcal{T}_\ell)$ from section 1.4.2 in each component of u, thus

$$(\phi_1, \ldots, \phi_{2|\mathcal{E}|}) = \big((\varphi_1, 0), \ldots, (\varphi_{|\mathcal{E}|}, 0), (0, \varphi_1), \ldots, (0, \varphi_{|\mathcal{E}|})\big)\,,$$

and the piecewise constant basis $\{\chi_m \mid 1 \leq m \leq |\mathcal{T}_\ell|\}$ of $P_0(\mathcal{T}_\ell)$ for p, which is defined for $T_m \in \mathcal{T}$ via $\chi_m \mid_{T_m} = 1$ and $\chi_m \mid_{T_j} = 0$ for $j \neq m$. We write u_ℓ and p_ℓ in terms of these basis functions as follows

$$u_\ell = \sum_{k=1}^{2|\mathcal{E}|} x_{u,k}\phi_k, \qquad p_\ell = \sum_{m=1}^{|\mathcal{T}|} x_{p,m}\chi_m$$

and introduce the index set

$$\mathcal{K} = \{k \in \{1, \dots, 2|\mathcal{E}|\} \mid \phi_k \text{ basis function of an interior edge}\}.$$

Then the equations (1.4.11) reduced to zero boundary conditions read

$$\begin{aligned}\sum_{k\in\mathcal{K}} x_{u,k} \int_\Omega \nabla_\ell\, \phi_k : \nabla_\ell\, \phi_j \,\mathrm{dx} - \sum_{m=1}^{|\mathcal{T}|} x_{p,m} \int_\Omega \chi_m \,\mathrm{div}_\ell\, \phi_j \,\mathrm{dx} \\ = \int_\Omega f \cdot \phi_j \,\mathrm{dx} + \int_{\partial\Omega} g \cdot \phi_j \,\mathrm{ds} - \sum_{k\in\{1,\dots,2|\mathcal{E}|\}\setminus\mathcal{K}} x_{u,k} \int_\Omega \nabla_\ell\, \phi_k : \nabla_\ell\, \phi_j \,\mathrm{dx}\end{aligned}$$

for all $j \in \mathcal{K}$,

$$\sum_{k\in\mathcal{K}} x_{u,k} \int_\Omega \chi_m \,\mathrm{div}_\ell\, \phi_k \,\mathrm{dx} + \lambda|T_m| = - \sum_{k\in\{1,\dots,2|\mathcal{E}|\}\setminus\mathcal{K}} x_{u,k} \int_\Omega \chi_m \,\mathrm{div}_\ell\, \phi_k \,\mathrm{dx}$$

for all $m = 1, \dots, |\mathcal{T}|$ as well as

$$\sum_{m=1}^{|\mathcal{T}|} x_{p,m}|T_m| = 0.$$

Defining the matrices $A \in \mathbb{R}^{2|\mathcal{E}|\times 2|\mathcal{E}|}$, $B \in \mathbb{R}^{|\mathcal{T}|\times 2|\mathcal{E}|}$ and $C \in \mathbb{R}^{1\times|\mathcal{T}|}$ via

$$\begin{aligned}A_{jk} &= \int_\Omega \nabla_\ell\, \phi_j : \nabla_\ell\, \phi_k \,\mathrm{dx}, \\ B_{mj} &= -\int_\Omega \chi_m \,\mathrm{div}_\ell\, \phi_j \,\mathrm{dx}, \\ C_m &= |T_m|\end{aligned}$$

for all $j, k = 1, \dots, 2|\mathcal{E}|$ and $m = 1, \dots, |\mathcal{T}|$ results in the linear system of equations

$$\begin{pmatrix} A & B^T & 0 \\ B & 0 & C^T \\ 0 & C & 0 \end{pmatrix} \begin{pmatrix} x_u \\ x_p \\ \lambda \end{pmatrix} = b,$$

which has to be solved on the interior edges.

The matrix A consists of two identical blocks A_{1D}, one for each component of u, and zero elsewhere.

$$A = \begin{pmatrix} A_{1D} & 0 \\ 0 & A_{1D} \end{pmatrix}.$$

These blocks can be computed and assembled as described in section 1.4.2. For the local stiffness matrices $B_T \in \mathbb{R}^{1\times 6}$ we use the fact that the vector containing $\operatorname{div} \phi_{j|T}$ is just the gradient matrix in a reordered form. The computation and elementwise storage of the local stiffness matrices is realised by

```
35  A4e=zeros(3,3,nrElems);
    B4e=zeros(1,6,nrElems);
    grads4e=zeros(3,2,nrElems);
    for j=1:nrElems
        grads=[c4n(n4e(j,:),:)'; 1 1 1]\[-2 0; 0 -2; 0 0];
40      A4e(:,:,j)=area4e(j)*(grads*grads');
        B4e(:,:,j)=-area4e(j)*grads(:)';
        grads=grads([3 1 2],:);
        grads4e(:,:,j) = grads;
    end
```

Then, the global stiffness matrix `A1D` and the block matrix `A` are assembled.

```
47  dofs_u=s4e(:,[2 3 1])';
    I=repmat(dofs_u(:),1,size(dofs_u,1))';
    J=repmat(dofs_u',1,size(dofs_u,1))';
50  A1D=sparse(I(:),J(:),A4e(:));
    A=[A1D sparse(nrSides,nrSides)
       sparse(nrSides,nrSides) A1D];
```

Analogously, `B` is assembled.

```
54  dofs_u=[s4e(:,[2 3 1]) nrSides+s4e(:,[2 3 1])];
55  dofs_p=1:nrElems;
    I=repmat(dofs_p(:),1,size(dofs_u,2))';
    J=repmat(dofs_u,1,size(dofs_p,1))';
    B=sparse(I(:),J(:),B4e(:),nrElems,2*nrSides);
```

The right-hand side vector `b` for homogeneous boundary conditions is computed using the midpoint quadrature rule.

```
61  mid4e=computeMid4e(c4n,n4e);
    f4e=f(mid4e).*[area4e area4e]/3;
    b4e=[f4e(:,1) f4e(:,1) f4e(:,1) f4e(:,2) f4e(:,2) f4e(:,2)];
    b=accumarray(dofs_u(:),b4e(:));
65  b=[b; zeros(nrElems+1,1)];
```

In order to include the given boundary conditions, one seeks the sides on the boundary and computes their length.

```
69  DbSides=zeros(1,size(n4sDb,1));
70  for j=1:size(n4sDb,1);
        DbSides(j)=s4n(n4sDb(j,1),n4sDb(j,2));
    end
    l4DbS=computeLength4s(c4n,n4sDb);
    freeSides=setdiff(1:size(n4s,1),DbSides);
```

Using the `integrate` function provided by AFEM the integral means of u_D along the boundary edges are computed.

```
76    mean4DbSides=integrate(c4n,n4sDb,@(x,y,z)(u4sDb(y)),10)./[l4DbS l4DbS];
```

Then, the modification of `b` for non-homogeneous Dirichlet boundary conditions reads

```
79    x=zeros(2*nrSides+nrElems+1,1);
80    x([DbSides nrSides+DbSides])=[mean4DbSides(:,1)' mean4DbSides(:,2)'];
      b=b-[A B' sparse(2*nrSides,1)]'*x(1:2*nrSides);
```

The Neumann boundary conditions are included via

```
84    for j=1:size(n4sNb,1)
85        side=s4n(n4sNb(j,1),n4sNb(j,2));
          b([side nrSides+side])=b([side nrSides+side]) +...
                                      length4s(side)*g(mid4s(side,:))';
      end
```

Again the midpoint quadrature approximates the integral of the boundary condition. Finally, the linear system is solved and one obtains the coefficient vectors for the velocity field and the pressure.

```
91    dofs=[freeSides nrSides+freeSides 2*nrSides+(1:nrElems) ...
                                                  2*nrSides+nrElems+1];
      M=[A B' sparse(2*nrSides,1);B sparse(nrElems,nrElems) area4e;...
                                        sparse(1,2*nrSides) area4e' 0];
95    x(dofs)=M(dofs,dofs)\b(dofs);
      nrDofs=length(dofs);
      u1=x(1:nrSides);
      u2=x(nrSides+1:2*nrSides);
      u=[u1 u2];
100   p=x(2*nrSides+1:2*nrSides+nrElems);
```

Optionally, the non-conforming gradient of u is computed and returned by

```
103   if nargout > 5
          gradU4e = zeros(2,2,nrElems);
105       for j = 1:nrElems
              gradU4e(1,:,j) = u1(s4e(j,:))' * grads4e(:,:,j);
              gradU4e(2,:,j) = u2(s4e(j,:))' * grads4e(:,:,j);
          end
      end
```

1.5 Estimate

In order to reduce the computational costs for a discrete solution with high accuracy adaptively refined meshes are helpful. These meshes should be automatically designed such that the resulting discrete solution has maximal accuracy compared to the size of the degrees of freedom. Therefore local refinement indicators resulting from global a posteriori error estimators are used to decide where the current triangulation of Ω shall be refined for the next level. For a deeper insight into this matter the reader is referred to [2, 5, 12].

1.5.1 Side-based Error Estimate for P_1

`[eta4s,n4s] = estimateP1EtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)`

The function `estimateP1EtaSides` implements a jump error estimator which is defined on each side. Reliability and efficiency are shown in [8]. The error for each side $S \in \mathcal{S}_\ell$ can be estimated by

$$\eta_\ell^2 := \sum_{S \in S_\ell} \eta_{\ell,S}^2, \quad \eta_{\ell,S}^2 := |S| \parallel [\nabla u_\ell]_S \cdot \nu_S \parallel_{L^2(S)}^2 .$$

Note that ∇u_ℓ is piecewise constant for $u_\ell \in V_C(\mathcal{T}_\ell)$, thus

$$\eta_{\ell,S}^2 = |S|^2 \, ([\nabla u_\ell]_S \cdot \nu_S)^2. \tag{1.5.1}$$

The implementation is realised in `estimateP1EtaSides`. First, the gradient of the discrete solution is computed on each element using the local gradients of the basis functions computed by (1.4.8).

```
25 for elem = 1 : nrElems
       grads = [1,1,1;c4n(n4e(elem,:),:)']\[0,0;eye(2)];
       gradU(elem,:) = x(n4e(elem,:))' * grads;
   end
```

`P0NormalJump` is used to compute the jumps on each side, which are then weighted with the corresponding length h_S, squared and returned in `eta4s`.

```
31 eta4s = (P0NormalJump(c4n,n4e,n4sDb,n4sNb,gradU,g).*length4s).^2;
```

1.5.2 Element-based Error Estimate for P_1

`[eta4e] = estimateP1EtaElements(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)`

Besides a purely side-based estimate, a volume term can be introduced:

$$\eta_\ell^2 := \sum_{T \in \mathcal{T}_\ell} |T| \|f\|_{L^2(T)}^2 + \sum_{S \in \mathcal{S}_\ell} \eta_{\ell,S}^2,$$

where $\eta_{\ell,S}^2$ is the error estimator from the previous Paragraph 1.5.1. In order to make this a purely element-based error estimate, let

$$\eta_{\ell,T}^2 := |T| \|f\|_{L^2(T)}^2 + \sum_{S \in S(T)} |S| \parallel [\nabla u_\ell]_S \cdot \nu_S \parallel_{L^2(S)}^2 .$$

The implementation of this estimator is realised in `estimateP1EtaElements`. First, $\eta_{\ell,S}$ is computed for each side.

```
23 eta4s = estimateP1EtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb);
```

In $\eta_{\ell,T}$ the L^2 norm of f is approximated by the midpoint quadrature rule

$$\|f\|_{L^2(T)}^2 \approx |T| f(\mathrm{mid}(T))^2.$$

This leads to the following Matlab implementation, which finally returns `eta4e`.

```
26 eta4e = (area4e.*f(mid4e)).^2 + sum(eta4s(s4e),2);
```

1.5.3 Averaging Error Estimate for P_1

```
eta4e = estimateP1AveragingP1(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
eta4e = estimateSigmaAveragingP1(c4n,n4e,sigma4e)
```

In order to estimate the error in the energy norm one may replace the unknown gradient of the exact solution by an averaged gradient using the averaging operator from Definition 1.8.2 on page 34. This approach results in the error indicator

$$\eta_\ell^2 := \sum_{T\in\mathcal{T}_\ell} \eta_{\ell,T}^2, \quad \eta_{\ell,T}^2 := \|A(\nabla_\ell u_\ell) - \nabla_\ell u_\ell\|_{L^2(T)}^2, \tag{1.5.2}$$

where ∇_ℓ is the piecewise gradient. The computation of the averaging operator is done in `estimateSigmaAveragingP1`. The final result is returned in `eta4e`.

```
for elem = 1 : nrElems
    grads = [1,1,1;c4n(n4e(elem,:),:)'] \ [0,0;eye(2)];
    grad4e(elem,:) = x(n4e(elem,:))'*grads;
end

eta4e = estimateSigmaAveragingP1(c4n,n4e,grad4e);
```

Let σ_ℓ (`sigma4e`) be a piecewise constant function on a triangulation $\mathcal{T}_\ell$ (`c4n`, `n4e`). The operator A is the averaging operator from Definition 1.8.2. First, the node values of $A(\sigma_\ell)$ are computed.

```
A = P0AveragingP1(c4n,n4e,sigma4e);
```

Note that evaluating an integrand in the midpoints of the sides of an element ensures exact quadrature up to polynomial degree two. Furthermore, $A(\sigma_\ell)$ is piecewise affine. For each $T \in \mathcal{T}_\ell$ the norm of the difference is computed in `estimateSigmaAveragingP1` by

$$\|A(\sigma_\ell) - \sigma_\ell\|_{L^2(T)}^2 = \frac{|T|}{3} \sum_{S\in\mathcal{S}(T)} \left(A(\sigma_\ell)(\mathrm{mid}(S)) - \sigma_\ell\right)^2,$$

which is implemented as follows.

```
for elem = 1 : nrElems
    s=0.5*[1 1 0;0 1 1;1 0 1]*A(n4e(elem,:),:)-[1;1;1]*sigma4e(elem,:);
    eta4e(elem) = sum(sum(s.^2))*area4e(elem)/3;
end
```

1.5.4 Side-based Error Estimate for *CR*

```
[eta4s,n4s] = estimateCREtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
```

In this implementation, a simplified version of the error estimate introduced in [10] is used. Proofs for reliability and efficiency can be found there. The omitted terms are of higher order. Let ∇_ℓ be the localised gradient on $\mathcal{T}_\ell$ and $u_\ell \in V_{CR}(\mathcal{T}_\ell)$ the discrete solution. Define

$$\eta_{\ell,CR}^2 := \sum_{S\in\mathcal{S}_{\ell,\Omega}\cup\mathcal{S}_{\ell,N}} |S| \|[\nabla_\ell u_\ell]_S \cdot \nu_S\|_{L^2(S)}^2 + \sum_{S\in\mathcal{S}_{\ell,\Omega}\cup\mathcal{S}_{\ell,D}} |S| \|[\nabla_\ell u_\ell]_S \cdot \tau_S\|_{L^2(S)}^2. \tag{1.5.3}$$

To estimate the discrete error a posteriori, (1.5.3) is implemented in `estimateCREtaSides`. First, the gradient of the discrete solution on each element is computed using the local gradients of the CR basis functions (cf. Section 1.4.2).

```
25 for elem = 1:size(n4e,1);
       grads = [c4n(n4e(elem,:),:)'; 1 1 1] \ [-2 0; 0 -2; 0 0];
       gradU(elem,:) = x(s4e(elem,:))' * grads([3 1 2],:);
   end
```

Afterwards, the jumps of the gradient in normal and tangential direction are computed using `P0NormalJump` and `P0TangentJump`, weighted with the corresponding length h_S and squared. The result is returned in `eta4s`.

```
31 eta4sNormal = P0NormalJump(c4n,n4e,n4sDb,n4sNb,gradU,g);
   eta4sTangent = P0TangentJump(c4n,n4e,n4sDb,n4sNb,gradU,u4Db);
   eta4s = ((eta4sNormal+eta4sTangent).*length4s).^2;
```

1.5.5 Side- and Volume-based Error Estimate for RT_0

`[eta4s,n4s] = estimateRT0EtaSides(f,g,u4Db,p,u,c4n,n4e,n4sDb,n4sNb)`

In this section, an error estimate for the mixed finite element method RT_0, based on sides as well as on a volume term, is introduced. For each side S, T_+ and T_- are defined as depicted in Figure 1.8.1 on page 33. Note that for boundary sides the number of T_- is set to 0. Define

$$\eta_\ell^2 := \sum_{S\in S_\ell} \eta_{\ell,S}^2, \quad \eta_{\ell,S}^2 := |S| \|[p_\ell]\cdot\tau_S\|^2 + \sum_{T_\pm\in\mathcal{T}_\ell} |T| \|f\|_T^2. \tag{1.5.4}$$

The implementation of this estimate can be found in `estimateRT0EtaSides` and needs the problem data `f`, `g`, and `u4Db`, the components of the computed solution `p` and `u` as well as the mesh data `c4n`, `n4e`, `n4sDb` and `n4sNb`. The norm of the jump of p across the sides is computed using the AFEM function `integrate` and `computeJump4s`.

```
35 jump4s = integrate(c4n, n4s, @(parts, Gpts4p, Gpt4ref)...
     computeJump4s(parts, Gpts4p, Gpt4ref, p, n4e, c4n,...
     e4s, s4Db, s4Nb, n4s, length4s, u4Db), 2);
```

The tangents of the sides and the midpoints of the elements are computed and the components of p are separated. For technical purposes the jump for boundary sides is computed with respect to an arbitrary element and is set to its correct value in lines 78-81.

```
58 tangent4s = computeTangent4s(c4n,n4parts);
   mid4e = computeMid4e(c4n, n4e);
60
   % components of the solution
   p1 = p(:,1);
   p2 = p(:,2);
   p3 = p(:,3);
65
   TPlus4s = e4s(:,1);
```

```
TMinus4s = e4s(:,2);

TMinus4s(TMinus4s == 0) = 1;
```

The solution is evaluated on each element with respect to the coefficients given by p and their respective basis functions.

```
valTPlus = sum ( (p1(TPlus4s) * [1 0] + p2(TPlus4s) * [0 1] ...
                  + (Gpts4p - mid4e(TPlus4s,:))...
          .* [p3(TPlus4s) p3(TPlus4s)]) .* tangent4s, 2);
valTMinus = sum ( (p1(TMinus4s) * [1 0] + p2(TMinus4s) * [0 1] ...
                  + (Gpts4p - mid4e(TMinus4s,:))...
          .* [p3(TMinus4s) p3(TMinus4s)]) .* tangent4s, 2);
```

The Dirichlet boundary condition is incorporated by $[\nabla u_D - p_\ell]\cdot\tau$, where $du_D/d\tau$ is computed by linear approximation. Hence the oscillation of u_D is required to be sufficiently small.

```
valTMinus(s4Db) = (u4Db(c4n(n4s(s4Db,2),:)) - u4Db(c4n(n4s(s4Db,1),:))) ...
                 ./ length4s(s4Db);
```

Finally, the jump across each side is computed and the values on the Neumann boundary are reset to 0, since the jump vanishes in normal direction, cf. (1.4.9).

```
jumps4s = (valTPlus - valTMinus) .^ 2;
jumps4s(s4Nb) = 0;
```

The area and the volume term for each element are evaluated.

```
vol4e = integrate(c4n, n4e, @(parts, Gpts4p, Gpt4ref) f(Gpts4p).^2, 2);

TPlus4s = e4s(:,1);
TMinus4s = e4s(:,2);
vol4eTMinus = zeros(size(n4s,1),1);
vol4eTMinus(TMinus4s(TMinus4s~=0)) = vol4e(TMinus4s(TMinus4s~=0));
area4eTMinus = zeros(size(n4s,1),1);
area4eTMinus(TMinus4s(TMinus4s~=0)) = area4e(TMinus4s(TMinus4s~=0));
```

The error is assembled by adding the jump and the volume term for each side according to (1.5.4) in the returned `eta4s`.

```
eta4s = length4s .* jump4s + area4e(TPlus4s).*vol4e(TPlus4s)/3 ...
       + area4eTMinus.*vol4eTMinus/3;
```

1.5.6 Side- and Volume-based Error Estimate for *CR* Solution of the Stokes' Equations

```
[eta4e,n4s] =
estimateNCStokesEtaElements(c4n,n4e,n4sDb,f,Du4Db1,Du4Db2,u,gradU4e)
```

The function `estimateNCStokesEtaElements` implements an error estimator for the non-conforming Crouzeix-Raviart solution of the Stokes' equations which is based on side and volume contributions. It reads

$$\eta_\ell^2 \coloneqq \sum_{T\in\mathcal{T}_\ell} \eta_\ell^2(T) \quad \text{with } \eta_\ell^2(T) \coloneqq |T|\|f\|_{L^2(\Omega)}^2 + |T|^{1/2} \sum_{S\in\mathcal{S}(T)} \|[\nabla_\ell u_\ell]_S \cdot \tau_S\|_{L^2(S)}^2$$

where ∇_ℓ denotes the piecewise gradient and τ_S the unit tangential vector on S with the orientation induced by T.

The input parameters are the usual geometric data `c4n`, `n4e`, `n4sDb` for the triangulation and the right-hand side function `f`. In order to include the Dirichlet boundary data the function `estimateNCStokesEtaElements` needs the two components of the gradient of u_D. Additionally the basis coefficients of the numerical solution of the velocity field with respect to the Crouzeix-Raviart basis described in section 1.4.4 is necessary. `gradU4e`, the piecewise derivative of the velocity field, is an optional parameter. As output, one obtains `eta4` $\in \mathbb{R}^{|\mathcal{T}|}$ which contains the values of $\eta_\ell^2(T)$ on each triangle $T \in \mathcal{T}$ and the matrix `n4s` for further computations.

If the optional input parameter is omitted, the gradients of the velocity field are computed as above. The computation of the jumps is performed elementwise with the advantage of the correct orientation of the tangent vectors.

```
44    jump4s=zeros(nrSides,2);
45    for j=1:nrElems
          jump4s(s4e(j,:),:) = jump4s(s4e(j,:),:) ...
                               + [tangent4e(:,:,j)*gradU4e(1,:,j)' ...
                                  tangent4e(:,:,j)*gradU4e(2,:,j)'];
      end
```

For the computation of the boundary jumps one has to consider the integral mean of the tangential derivative of the boundary function.

```
52    l4DbS=computeLength4s(c4n,n4sDb);
      mean4DbSides1=integrate(c4n,n4sDb,@(x,y,z)(Du4Db1(y)),10)...
                                                    ./[l4DbS l4DbS];
55    mean4DbSides2=integrate(c4n,n4sDb,@(x,y,z)(Du4Db2(y)),10)...
                                                    ./[l4DbS l4DbS];
      for j=1:size(n4sDb,1)
          nodes=n4sDb(j,:)';
          side=s4n(nodes(1),nodes(2));
60        jump4s(side,:) = jump4s(side,:) ...
                           - [mean4DbSides1(j,:)*tangent4s(side,:)'...
                              mean4DbSides2(j,:)*tangent4s(side,:)'];
      end
```

where `computeLength4s`, `computeTangent4s` and `integrate` are AFEM routines. Finally, `eta4e` is assembled by the formula mentioned above.

```
66    jump4s_L2normSq=(jump4s(:,1).^2 + jump4s(:,2).^2) .* length4s;
      [~,L2normSq4e]=L2Norm(c4n,n4e,@(x,y,z)f(y),6);
      eta4e = area4e.*sum(L2normSq4e,2) +...
                              sqrt(area4e).*sum(jump4s_L2normSq(s4e),2);
```

1.6 Mark

In step Mark of the AFEM cycle, sides are selected for refinement. Each marking algorithm takes a list of elements or sides (abbreviated by `p` for parts) of the current mesh, along with refinement indicators for each part as input.

The list of elements or sides is given by `n4e` or `n4s` as a $|\mathcal{T}_\ell| \times 3$ matrix or a $|\mathcal{S}_\ell| \times 2$ matrix, respectively. Each row consists of the nodes of the corresponding element or side. The refinement indicators `eta4e` or `eta4s` are given as a $|\mathcal{T}_\ell|$- or $|\mathcal{S}_\ell|$-dimensional column vector.

The output of all marking algorithms is allways a list of marked sides, regardless of the input. This list follows the same pattern as `n4s`, i.e., it consists of one row for each marked side where the side is given by its end nodes. If `n4e` was the input, all sides of any marked element will be marked.

Three different criteria are available in AFEM, namely uniform and two different adaptive refinements. Let

$$\eta^2_{\ell,T}(T_{|\mathcal{T}_\ell|}) \leq \cdots \leq \eta^2_{\ell,T}(T_1),$$
$$\eta^2_{\ell,S}(S_{|\mathcal{S}_\ell|}) \leq \cdots \leq \eta^2_{\ell,S}(S_1),$$

be the refinement indicators for the elements $T_1, \ldots, T_{|\mathcal{T}_\ell|}$ or sides $S_1, \ldots, S_{|\mathcal{S}_\ell|}$ and

$$\eta^2_{\ell,\mathcal{T}} := \sum_{T \in \mathcal{T}_\ell} \eta^2_{\ell,T}(T)$$
$$\eta^2_{\ell,\mathcal{S}} := \sum_{S \in \mathcal{S}_\ell} \eta^2_{\ell,S}(S)$$

- Mark for uniform refinement
  `n4sMarked = markUniform(n4p)`

When using `markUniform`, all sides are marked.

- Maximum Criterion
  `n4sMarked = markMaximum(n4p,eta4p,OPTtheta)`

In `markMaximum` those sides are marked for which the refinement indicator $\eta^2_{\ell,\mathcal{T}}$ or $\eta^2_{\ell,\mathcal{S}}$, given by `eta4e` or `eta4s`, respectively, is greater or equal than θ times the largest occuring value $\eta^2_{\ell,T}(T_1)$ or $\eta^2_{\ell,S}(S_1)$. The set $\mathcal{M}_\ell$ of marked sides is

$$\mathcal{M}_\ell := \begin{cases} \{S_j \in \mathcal{S}_\ell \mid \theta\eta^2_{\ell,S}(S_1) \leq \eta^2_{\ell,S}(S_j)\} & \text{for } \texttt{n4s} \text{ as input} \\ \{S \in \mathcal{S}(T_j) \mid \theta\eta^2_{\ell,T}(T_1) \leq \eta^2_{\ell,T}(T_j),\ T_j \in \mathcal{T}_\ell\} & \text{for } \texttt{n4e} \text{ as input.} \end{cases}$$

This set is collected in `n4sMarked` and returned. The parameter `OPTtheta` is optional. Its default value is $\theta = 1/2$.

- Bulk Criterion
  `n4sMarked = markBulk(n4p,eta4p,OPTtheta)`

In `markBulk` the marking algorithm will gradually mark the sides with the largest values of the refinement indicator. The set of marked sides is defined as a set of minimal

cardinality

$$\mathcal{M}_\ell := \begin{cases} \{S_1, \dots, S_k\} \text{ with } \theta\eta_{\ell,\mathcal{S}}^2 \le \sum_{j=1}^{k} \eta_{\ell,S}^2(S_j), & \text{for } \texttt{n4e} \text{ as input} \\ \bigcup_{1\le j\le k} \mathcal{S}(T_j) \text{ with } \theta\eta_{\ell,\mathcal{T}}^2 \le \sum_{j=1}^{k} \eta_{\ell,T}^2(T_j) & \text{for } \texttt{n4s} \text{ as input.} \end{cases}$$

The parameter `OPTtheta` is optional. By default, θ is set to $1/2$.

1.7 Refine

In step Refine of the AFEM cycle the mesh is refined with respect to the marked parts.

The initial mesh $\mathcal{T}_0$ is a regular triangulation of $\Omega \subset \mathbb{R}^n$ into closed triangles in the sense of Definition 1.3.2. We assume that each element in $\mathcal{T}_0$ has at least one vertex in the interior of Ω.

Given any $T \in \mathcal{T}_0$, one of its sides $\mathcal{S}(T)$ is chosen to be its reference side $S(T)$.

1.7.1 Closure Algorithm

```
n4sRefine = closure(n4e,n4sMarked)
```

Most of the refinement methods rely on the closure algorithm in order to avoid degenerated meshes.

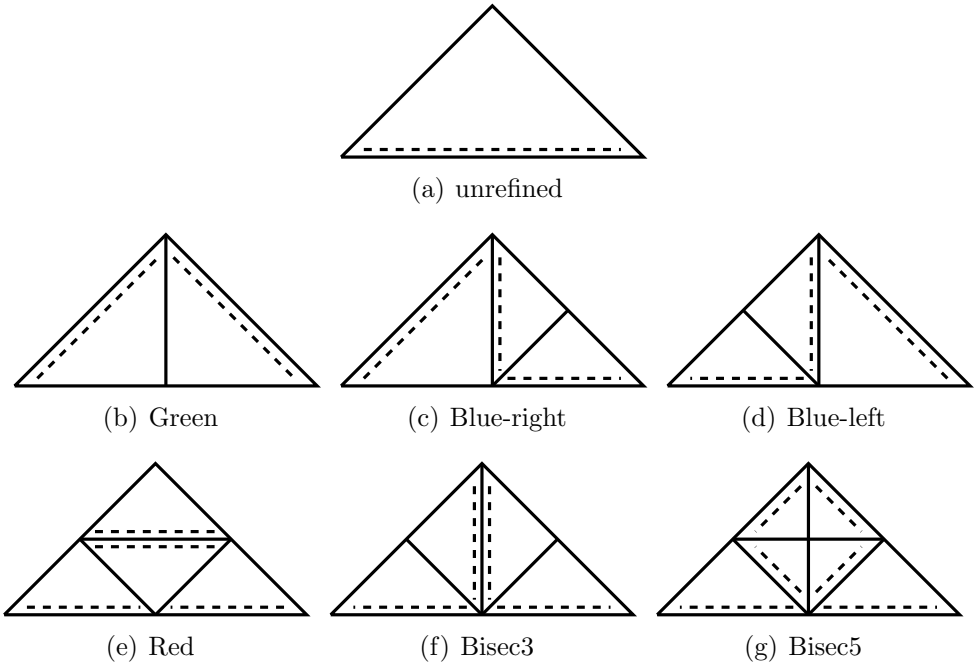
Given the mesh structure (`n4e`) and a list of marked sides (`n4sMarked`, cf. `n4s`), `closure` computes a list of sides (`n4sRefine`) which have to be refined such that whenever a side of an element T is marked for refinement, its reference side $S(T)$ is marked as well.

In any refinement method that relies on the closure algorithm, `closure` is called within this method. In general, there is no need to explicitly call `closure` outside of the refinement algorithms.

1.7.2 Refining Algorithms

This subsection is devoted to a short review of the refinement algorithms that are already included in AFEM. All refinement methods, besides `refineUniformRed`, refine the given mesh according to the set of marked sides. Once $T \in \mathcal{T}_\ell$ is refined, the reference sides will be specified for the subtriangles as indicated in Figure 1.7.1, where the reference sides of the unrefined element and of all the new elements are marked by dashed lines. Whenever a side is refined, it is bisected and the new node is the midpoint of the original side.

Figure 1.7.1: The reference element and its possible refinements.



(a) unrefined

(b) Green (c) Blue-right (d) Blue-left

(e) Red (f) Bisec3 (g) Bisec5

• Red refinement

```
[c4nNew,n4eNew,n4sDbNew,n4sNbNew] = ...
        refineUniformRed(c4n,n4e,n4sDb,n4sNb)
```

`refineUniformRed` is a refinement algorithm that does not rely upon marked sides. It merely refines every element of the mesh given by `c4n` and `n4e` using the Red strategy (see Figure 1.7.1(e)). It returns the updated data structures `c4n`, `n4e`, `n4sDb` and `n4sNb`.

• Red-Green-Blue refinement

```
[c4nNew,n4eNew,n4sDbNew,n4sNbNew] =
        refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked)
```

`refineRGB` first runs the closure algorithm and then refines each element according to its marked sides using either the Red, Green or Blue refinements. Owing to `closure`, for any element with at least one marked side the reference side of that element is marked as well. Thus, one of the following three cases applies.

i. If *only one side* (i.e., the reference side) of an element is marked, the Green refinement strategy is used for that triangle.

ii. If *two sides* (i.e., the reference side and one other side) of an element are marked, then one of the Blue strategies is chosen to refine this element. If the second marked side is the second side in the corresponding row of `n4e`, then Blue-right will be used, otherwise Blue-left is applied.

iii. If *all three sides* of an element are marked, then the Red strategy is used for refinement of that element.

• Bisec3-Green-Blue and Bisec5-Green-Blue refinement

The refinement algorithms used in `refineBi3GB` and `refineBi5GB` are analogous to `refineRGB` as described above, besides in the case that all three sides are marked for refinement. Whenever `refineRGB` uses the Red strategy, `refineBi3GB` and `refineBi5GB` apply the Bisec3 and Bisec5 strategies, respectively. This ensures the methods to be equivalent to the local NVB (Newest-Vertex Bisection) rule described in [8, 6].

1.7.3 Guaranteed Properties of the Triangulations

This subsection lists some properties of a triangulation $\mathcal{T}_\ell$ obtained by the refinement algorithms described above and under the assumptions on $\mathcal{T}_0$ from the beginning of this section.For the non-elementary proofs the reader is referred to [9].

An element $T \in \mathcal{T}_0$ is called *isolated* if its reference side is not the reference side of any other element in $\mathcal{T}_\ell$. Given a regular triangulation $\mathcal{T}_0$, Algorithm 2.1 of [9] provides a distribution of reference sides $S(T)$ for all $T \in \mathcal{T}_0$ such that two distinct isolated triangles do not share a side. This is important for the H^1 stability of the L^2 projection as follows.

Given positive constants C_{stab} and C_{app} depending exclusively on the initial triangulation $\mathcal{T}_0$, the following properties hold.

i. $\mathcal{T}_\ell$ is a regular triangulation of Ω into triangles. For each $T \in \mathcal{T}_\ell$ there exists one reference side $S(T)$ which depends only on T but not on the level ℓ.

ii. For each $K \in \mathcal{T}_0$, $\mathcal{T}_\ell|_K := \{T \in \mathcal{T}_\ell \,|\, T \subseteq K\}$ is the image under an affine map $\Phi : K \to T_{\text{ref}}$ onto the reference triangle $T_{\text{ref}} = \text{conv}\{(0,0),(0,1),(1,0)\}$ with $\Phi(S(K)) = \text{conv}\{(0,0),(1,0)\}$ and $\det D\Phi > 0$. The triangulation

$$\widehat{\mathcal{T}}_K := \{\Phi(T) : T \in \mathcal{T}, T \subseteq K\}$$

of K consists of right isosceles triangles. Recall that a right isosceles triangle results from a square halved along a diagonal.

iii. The L^2 projection Π onto $V_\ell := P_1(\mathcal{T}_\ell) \cap V$ is H^1 stable. The piecewise affine spaces are defined by

$$\begin{aligned} P_1(T) &:= \{v \in C^\infty(T) : v \text{ affine on } T\}, \\ P_1(\mathcal{T}_\ell) &:= \{v \in L^\infty(\Omega) : \forall T \in \mathcal{T}_\ell, v|_T \in P_1(T)\}. \end{aligned}$$

For any $v \in V := H^1_0(\Omega)$ the L^2 projection Πv on V_ℓ satisfies

$$\|\nabla \Pi v\|_{L^2(\Omega)} \leq C_{\text{stab}} \|\nabla v\|_{L^2(\Omega)}.$$

iv. The approximation property of the L^2 projection states

$$\sum_{T\in\mathcal{T}_\ell} \|h_T^{-1}(v-\Pi v)\|^2_{L^2(T)} + \sum_{E\in\mathcal{E}_\ell} \|h_E^{-1/2}(v-\Pi v)\|^2_{L^2(E)} \leq C_{\mathrm{app}} \|\nabla v\|^2_{L^2(\Omega)}$$

for all $v \in V$.

1.8 Mathematical Notations

1.8.1 Jumps and Averaging

A variety of error estimators use the jump or smoothing techniques of the discrete gradient for diverse finite element methods. Therefore, some basic functions computing jumps and smoothings of piecewise constant functions are included in AFEM to assist the user in realising new error estimators in less time.

Definition 1.8.1. For two neighbouring triangles $T_+, T_- \in \mathcal{T}_\ell$ with a common side $S \in \mathcal{S}_{\ell,\Omega}$, the jump of a vector-valued function $\sigma_\ell : \mathbb{R}^2 \to \mathbb{R}^2$ across S is given by $[\sigma_\ell]_S := \sigma_{|T_+} - \sigma_{|T_-}$. On boundary sides, the conditions of the model problem can be used to compute the jump in tangential direction on the Dirichlet boundary and in normal direction on the Neumann boundary. The jump can be decomposed into the jumps in normal and tangential direction. The jump of σ_ℓ in normal direction on a side $S \in \mathcal{S}_\ell \setminus \mathcal{S}_{\ell,D}$ is given by

$$[\sigma_\ell]_S \cdot \nu_S := \begin{cases} \sigma_\ell\big|_{T_+} \cdot \nu_S - \sigma_\ell\big|_{T_-} \cdot \nu_S & \text{for } S \in \mathcal{S}_{\ell,\Omega}, \\ \sigma_\ell\big|_{T_+} \cdot \nu_S - g & \text{for } S \in \mathcal{S}_{\ell,N}, \end{cases} \tag{1.8.1}$$

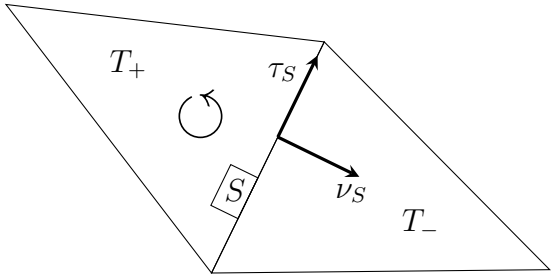
where ν_S is the unit normal vector on S pointing from T_+ to T_- as depicted in Figure 1.8.1. In AFEM the enumeration introduced in `e4s` induces the triangle T_+ for each side. The jump of σ_ℓ in tangential direction on a side $S \in \mathcal{S}_\ell \setminus \mathcal{S}_{\ell,N}$ is given by

$$[\sigma_\ell]_S \cdot \tau_S := \begin{cases} \sigma_\ell\big|_{T_+} \cdot \tau_S - \sigma_\ell\big|_{T_-} \cdot \tau_S & \text{for } S \in \mathcal{S}_{\ell,\Omega}, \\ \sigma_\ell\big|_{T_+} \cdot \tau_S - \nabla u_D \cdot \tau_S & \text{for } S \in \mathcal{S}_{\ell,D}, \end{cases} \tag{1.8.2}$$

where τ_S is the unit tangent vector on S pointing in counterclockwise direction w.r.t. T_+ as in Figure 1.8.1.

The methods introduced in the sequel measure the jump of the gradient of the discrete solution. The gradient of piecewise P_1 functions is constant on each element. The following functions compute the jump of any piecewise P_0 function.

Figure 1.8.1: Unit normal vector ν_S and unit tangent vector τ_S.



- Jump of a P_0 function in normal direction across sides
  `jump4s = P0NormalJump(c4n,n4e,n4sDb,n4sNb,sigma4e,g)`

`P0NormalJump` computes the jump in normal direction of a piecewise constant function $\sigma : \mathbb{R}^2 \to \mathbb{R}^2$, given by `sigma4e`, across each side of a triangulation given by `c4n`, `n4e`, `n4sDb` and `n4sNb`. The Neumann boundary data for the sides in $\mathcal{S}_{\ell,N}$ are given by `g`.

The first step sums up the function values in outer normal direction w.r.t. each element for each side. Thus, for the inner sides the two added values contribute to the jump in normal direction, while for the boundary sides only one function value in outer normal direction is available.

```
19 for elem = 1 : size(n4e,1)
20     jump4s(s4e(elem,:)) = jump4s(s4e(elem,:)) ...
                           + normal4e(:,:,elem)*sigma4e(elem,:)';
   end
```

The value of `g` is subtracted for Neumann boundary sides in a second step.

```
25 for nodes = n4sNb'
       side = s4n(nodes(1),nodes(2));
       jump4s(side) = jump4s(side) - g(mid4s(side,:));
   end
```

On Dirichlet boundary sides no condition on the outer unit normal direction is available, thus the jump is reset to 0.

```
31 jump4s(diag(s4n(n4sDb(:,1),n4sDb(:,2)))) = 0;
```

The last step ensures that the jump values are non-negative.

```
33 jump4s = abs(jump4s);
```

- Jump of a P_0 function in tangential direction across sides
  `jump4s = P0TangentJump(c4n,n4e,n4sDb,n4sNb,sigma4e,u4Db)`

`P0TangentJump` computes the jump in tangential direction of a piecewise constant function $\sigma : \mathbb{R}^2 \to \mathbb{R}^2$, given by `sigma4e`, across each side of a triangulation given by `c4n`, `n4e`, `n4sDb` and `n4sNb`. The Dirichlet boundary data for the sides in $\mathcal{S}_{\ell,D}$ are given by `u4Db`. At first, the function values in tangential direction in counterclockwise orientation w.r.t. each element are summed up for each side. Thus, for the inner sides the two added

values contribute to the jump in tangential direction, while for the boundary sides only one function value in tangential direction is available.

```
for elem = 1 : size(n4e,1)
    jump4s(s4e(elem,:)) = jump4s(s4e(elem,:)) ...
                        + tangent4e(:,:,elem)*sigma4e(elem,:)';
end
```

On Neumann boundary sides there is no condition for the behaviour in tangential direction, thus the jump is reset to 0.

```
jump4s(diag(s4n(n4sNb(:,1),n4sNb(:,2)))) = 0;
```

On Dirichlet boundary sides the jump is calculated by subtracting the derivative of the Dirichlet boundary data given by `u4Db` from the already considered function value in tangential direction. The derivative of the Dirichlet boundary data is approximated using finite differences.

```
for nodes = n4sDb'
    side = s4n(nodes(1),nodes(2));
    jump4s(side) = jump4s(side) ...
        - (u4Db(c4n(nodes(2),:))-u4Db(c4n(nodes(1),:)))/length4s(side);
end
```

The last step ensures that the jump values are non-negative.

```
jump4s = abs(jump4s);
```

Another important error estimator utilises the following averaging operator.

Definition 1.8.2. For a triangulation $\mathcal{T}_\ell$ of Ω with nodes $z \in \mathcal{N}_\ell$ and a piecewise constant function $\sigma : \Omega \to \mathbb{R}^d$, define the *averaging operator* $A(\sigma)$ as the piecewise P_1 interpolation

$$A(\sigma) := \sum_{z\in\mathcal{N}_\ell} A(\sigma)(z)\varphi_z \quad \text{with } A(\sigma)(z) := \sum_{\substack{T\in\mathcal{T}_\ell \\ T\subseteq\omega_z}} \frac{|T|}{|\omega_z|}\sigma(\mathrm{mid}(T)),$$

where ω_z is the node patch of z.

- Averaging for a piecewise constant function
  `val = P0AveragingP1(c4n,n4e,sigma4e)`

Given a triangulation (`c4n`, `n4e`) and the values of an elementwise constant function $\sigma : \Omega \to \mathbb{R}^d$ (`sigma4e`, a $|\mathcal{T}_\ell| \times d$ matrix), in `P0AveragingP1` the node values of $A(\sigma)$ as defined in Definition 1.8.2 are computed. First, the values of σ are weighted with the corresponding $|T|$. The major part of the work is done by accumulation, where the weighted values of σ are added up for each node patch. Finally, these values are divided by $|\omega_z|$.

```
weightedV = sigma4e.*(area4e*ones(1,d))*[eye(d),eye(d),eye(d)];
I = n4e(:)*ones(1,d);
J = ones(3*size(n4e,1),1)*(1:d);
val = accumarray([I(:),J(:)],weightedV(:))./(area4n*ones(1,d));
```

1.8.2 Numerical Quadrature

Several methods in the AFEM package need to integrate functions on 1D or 2D domains numerically, for example to compute the right-hand side of the weak form

$$\int_\Omega f\varphi \, dx + \int_{\Gamma_N} g\varphi \, dx.$$

• Integration Interface
`val = integrate(c4n, n4p, integrand, degree, OPTsize4parts)`

The method `integrate` provides a unified interface to integrate functions with arbitrary degree of accuracy. For integration over a given domain in 1D or 2D, a partition into elements or sides described by coordinates for nodes `c4n` and nodes for parts `n4p`, has to be specified. Here, `n4p` consists of a subset of rows of either `n4e` or `n4s`. The integrand `integrand`, defined by a vector or matrix valued function $f : \mathbb{R}^r \to \mathbb{R}^{s\times t}$, is described below. The value `degree` determines the order of the quadrature formula such that polynomials up to specified total degree are integrated exactly. The optional parameter `OPTsize4parts` includes `length4s` for 1D domains or `area4e` for 2D domains. The output `val` is a three dimensional matrix of size `[nrParts s t]` where `nrParts = size(n4p,1)`.
The integrand is passed to the interface through a user defined function handle `integrand` of the following form

```
function val = integrand(n4p,Gpts4p,Gpt4ref).
```

Here `n4p` is the same as above, `Gpts4p` holds the coordinates of the current Gauss point for each part whereas `Gpt4ref` is that Gauss point on the reference part (i.e. the standard 1D or 2D simplex with the nodes $(0), (1)$ or $(0,0), (0,1), (1,0)$). The output `val` of `integrand` contains its evaluation at the given Gauss point for each part in `n4p`. In the sequel the quadrature formulas for 1D and 2D integrals implemented in inline functions are described.

• Gauß-Legendre Formula
`[x,w] = getGaussPoints(n)`

In order to integrate a function f over a side S, the 1D Gauß-Legendre quadrature formula

$$\int_S f(x) \, dx \approx \sum_{k=1}^{n} \omega_k f(x_k)$$

is employed with positive $n \in \mathbb{N}$, weights ω_k and Gauß points x_k, $k = 1, \ldots, n$. The inline function `getGaussPoints` computes the weights and points for the reference interval $[0, 1]$ such that the resulting quadrature formula is exact for all polynomials up to degree $p = 2n-1$. This choice is optimal w.r.t. the number of points and weights. For the interval $[-1, 1]$, the Gauß points are the roots of the n-th Legendre polynomial. The following properties show an efficient method to calculate the Gauß points x_k and weights ω_k for arbitrary positive $n \in \mathbb{N}$.

- The roots $x_k,\ k = 1, \ldots, n$ of the n-th Legendre polynomial p_n are the eigenvalues of the tridiagonal matrix

$$J_n := \begin{bmatrix} \delta_1 & \gamma_1 & & & \\ \gamma_1 & \delta_2 & \cdot & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \gamma_{n-1} \\ & & & \gamma_{n-1} & \delta_n \end{bmatrix},$$

where

$$\delta_m := 0 \quad \text{for } 1 \le m \le n \qquad \text{and} \qquad \gamma_m := \frac{m}{\sqrt{4m^2 - 1}} \quad \text{for } 1 \le m \le n-1.$$

The Gauß points x_k are computed in the following Matlab lines.

```
98  gamma = [1 : n-1] ./ sqrt(4*[1 : n-1].^2 - ones(1,n-1) );
    [V,D] = eig( diag(gamma,1) + diag(gamma,-1) );
100 x = diag(D);
```

- For the eigenvector $v^{(k)}$ belonging to the eigenvalue x_k of J_n, the weight ω_k is set to $2\left(v_1^{(k)}\right)^2$ for $k = 1, \ldots, n$.

Finally, the interval $[-1, 1]$ is transformed to the reference interval $[0, 1]$.

• Conical-Product Formula
`[x,w] = getConProdGaussPoints(n)`

In order to integrate a given function f over a triangle, the interface `integrand` uses the Conical-Product formula which is a combination of two 1D-Gauß formulas. The in-line function `getConProdGaussPoints` computes the n^2 points x_k and weights w_k for the reference triangle T_{ref}. The composite formula for the quadrangle is simply the cartesian product of the 1D-Gauß-Legendre formula. To get a formula for a triangle, the quadrangle is transformed to a triangle by the Duffy-transformation, cf. [11]. For $y_1, y_2 \in [0, 1]^2$ this transformation reads

$$\begin{aligned} z_1 &= y_1, \\ z_2 &= y_2(1 - y_1) = y_2(1 - z_1)\ . \end{aligned}$$

Since $0 \le y_i \le 1,\ i = 1, 2$, we conclude $0 \le z_1 \le 1,\ 0 \le z_2 \le 1 - z_1$, i.e., $z = (z_1, z_2) \in T_{ref}$. Thus, the Conical-Product formula is a combination of the 1D-Gauß formulas for the integrals

$$\int_0^1 f(r)\ dr \approx \sum_{j=1}^{n} a_j f(r_j)\ ,$$

$$\int_0^1 (1 - s) f(s)\ ds \approx \sum_{k=1}^{n} b_k f(s_k)\ .$$

The first integral can be calculated as above with the Gauß-Legendre formula. For the second integral, a quadrature formula for weighted functions $\omega(x)f(x)$ has to be used. Since the weight function here is $\omega(x) = (1-x)$, the Gauß-Jacobi formula is used. Therefore, the order of the method is preserved, namely n^2 points lead to exact integration of polynomials up to total degree $p = 2n - 1$.

Note that also choosing the Gauß-Legendre formula for the second integral results in integration of less accuracy. The roots of the corresponding Jacobi polynomials can be calculated analogously to the roots of the Gauß-Legendre polynomials. The coefficients for the Jacobi polynomials are

$$\delta_m := \frac{-1}{4m^2-1} \quad \text{for } 1 \le m \le n \quad \text{and} \quad \gamma_m := \sqrt{\frac{m(m+1)}{2(m+1)-1}} \quad \text{for } 1 \le m \le n-1.$$

```
126 delta = -1./(4*(1 : n).^2-ones(1,n));
    gamma = sqrt((2 : n).*(1 : n-1)) ./ (2*(2 : n)-ones(1,n-1));
    [V,D] = eig( diag(delta)+diag(gamma,1)+diag(gamma,-1) );
    s = diag(D);
130 b = 2*V(1,:).^2;
```

Here, the weights are scaled with respect to $\int_{-1}^{1}(1-s)ds = 2$. Again, the Gauß points are transformed to the interval $[0,1]$. Since $\int_{0}^{1}(1-s)ds = 1/2$, the weights of the Gauß-Jacobi formula are divided by 4. Hence, the Conical-Product formula uses the n^2 Gauß points $(s_k, r_j(1-s_k))$ and weights $a_j b_k$ for $1 \le j,k \le n$.

```
141 s = repmat(s',n,1); s = s(:);
    r = repmat(r,n,1);
    x = [ s , r.*(ones(n^2,1)-s) ];
    w = a*b';
145 w = w(:);
```

1.8.3 L^2 Norm and Oscillations

- L^2 Norm
  `[L2norm4Omega, L2norm4p] = L2Norm(c4n,n4e,f,degree)`

The necessity of computing the L_2 norm of a function on a triangulation (`c4n`, `n4e`) arises frequently in the context of AFEM. `L2Norm` does this for functions `f` which satisfy the input/output behaviour demanded by `integrate` using a quadrature formula which is exact up to degree `degree`. `L2Norm4Omega` and `L2Norm4p` are returned.

```
11 L2norm4p = integrate(c4n, n4e, ...
             @(n4p,Gpts4p,Gpts4ref)(f(n4p,Gpts4p,Gpts4ref)).^2, degree);
   L2norm4Omega = sqrt(sum(L2norm4p));
```

- L^2 Error for P_1
  `error4e = error4eP1L2(c4n,n4e,uExact,uApprox)`

In order to measure the quality of a given numerical solution u_ℓ one can use the L^2 error between the given exact solution `val = uExact(x)` and the computed solution `uApprox` as returned by `solveP1Poisson`. For each element $||u - u_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for P_1 Gradient
  `error4e = error4eP1Energy(c4n,n4e,gradExact,uApprox)`

In order to measure the quality of a given numerical solution u_ℓ one can use the L^2 error between the given exact gradient `val = gradExact(x)` and the gradient of the computed solution `uApprox` as returned by `solveP1Poisson`. For each element $||\nabla u - \nabla u_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for CR
  `error4e = error4eCRL2(c4n,n4e,uExact,uApprox)`

In order to measure the quality of a given numerical solution u_ℓ one can use the L^2 error between the given exact solution `val = uExact(x)` and the computed solution `uApprox` as returned by `solveCRPoisson`. For each element $||u - u_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for CR Gradient
  `error4e = error4eCREnergy(c4n,n4e,gradExact,uApprox)`

In order to measure the quality of a given numerical solution u_ℓ one can use the L^2 error between the given exact gradient `val = gradExact(x)` and the gradient of the computed solution `uApprox` as returned by `solveCRPoisson`. For each element $||\nabla u - \nabla u_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for RT_0
  `error4e = error4eRT0L2(c4n,n4e,uExact,uApprox)`

In order to measure the quality of a given numerical solution u_ℓ one can use the L^2 error between the given exact solution `val = uExact(x)` and the computed solution of the first component `uApprox` as returned by `solveRT0Poisson`. For each element $||u - u_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for RT_0 Gradient
  `error4e = error4eRT0Energy(c4n,n4e,gradExact,pApprox)`

In order to measure the quality of a given numerical solution p_ℓ one can use the L^2 error between the given exact gradient `val = gradExact(x)` and the gradient component

of the computed solution `pApprox` as returned by `solveRT0Poisson`. For each element $||p - p_\ell||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately.

- L^2 Error for Stress Tensor of a CR-P_0 Solution of the Stokes' Equations
  `error4eStokesCRStress(c4n,n4e,component,stressExact,...`
  `uApprox,pApprox,gradUApprox)`

In order to measure the quality of a given numerical solution u_ℓ and p_ℓ of the Stokes' equations, which gives the discrete pseudostress $\sigma_\ell := \nabla u_\ell - p_\ell I_{2\times 2} \in P_0(\Omega; \mathbb{R}^{2\times 2})$, one can use the L^2 error between the given exact pseudostress tensor `val = stressExact` and the discrete pseudostress of the computed solution `uApprox` and `pApprox` as returned by `solveCRP0Stokes`. This function only works for one line vector and thus have to be applied separately to each of the two components of the tensor. The component number $j = 1, 2$ has to be passed as the argument variable `component`. For each element $||\sigma(j,:) - \sigma_\ell(j,:)||^2_{L^2(T)}$, $T \in \mathcal{T}_\ell$ is returned approximately. For better performance the corresponding gradient component can be passed as an optional argument.

- Oscillations
  `[osc4e, mean4e] = oscillations(c4n,n4e,f,degree)`

If the right-hand side f of the problem (1.2.1) or (1.2.3) oscillates strongly, this effect should be taken into account when refining a mesh (e.g., in error estimators). For $f \in L^2(\Omega)$ the oscillation of f w.r.t. a triangulation $\mathcal{T}_\ell$ is given by

$$\operatorname{osc}_\ell^2 := \operatorname{osc}_\ell^2(f, \mathcal{T}_\ell) := \sum_{T\in\mathcal{T}_\ell} \operatorname{osc}^2(f,T), \text{ with}$$
$$\operatorname{osc}(f,T) := \|f - f_T\|_{L^2(T)},$$

where f_T denotes the integral mean of f on a triangle T

$$f_T := \fint_T f dx := \int_T f dx / |T|.$$

In `oscillations`, the $\operatorname{osc}_\ell^2(f,T)$ of `f` is computed on each element of the triangulation given by `c4n` and `n4e`. The integral mean is accurate for polynomials up to degree `degree`. The integral in the L^2 norm is approximated with a quadrature formula which is accurate up to 2∗`degree`. Oscillations are computed using `integrate`.

```
9     meanDeg = degree;
10    oscDeg = 2*degree;
      area4e = computeArea4e(c4n,n4e);

      mean4e = integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)(f(Gpts4p)),meanDeg)...
               ./area4e;
15
      osc4e = integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)((f(Gpts4p)-mean4e).^2),...
              oscDeg);
      osc4e = osc4e .* area4e;
```

1.9 Utility Functions

In order to allow the reader to focus on creating new solvers and error estimators various common tasks are accomplished by numerous utility functions referenced in this section.

1.9.1 Enumeration Functions

In addition to the basic geometry data further data structures can be easily computed by simply applying the functions described below.

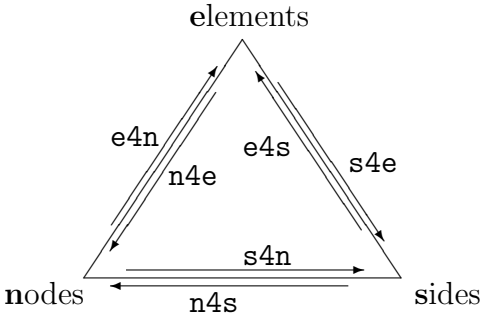
Nodes and elements are numbered as they occur in `c4n`and `n4e`. As these two data structures impose no natural numbering of the sides, an arbitrary (but fixed) numbering is chosen. This initial side numbering remains the same for all structures depending on it.

Note that some functions generate sparse matrices. For those, the sparsity constant is bounded due to the used mesh generation.

Mesh Structure Data

The mesh structure data provides additional information about the mesh structure given by `n4e`. As this data does not depend on the exact location of the nodes, the only input is `n4e`. The effect of these functions is illustrated in Figure 1.9.1.

Figure 1.9.1: Relations between elements (`e`), sides (`s`) and nodes (`n`).



• Load a geometry
`[c4n n4e n4sDb n4sNb] = loadGeometry(name, OPTRefinementLevel)`

`loadGeometry` loads the mesh data of the geometry `name`. It returns the data structures `c4n`, `n4e`, `n4sDb`and `n4sNb`. Optionally, the second parameter `OPTRefinementLevel` will cause the mesh to be refined using the uniform red strategy the given number of times. Its default value is set to 0. The files from which the data is loaded are required to be named "`name`_c4n.dat", "`name`_n4e.dat", "`name`_n4sDb.dat" and "`name`_n4sNb.dat".

- Elements for nodes
  `e4n = computeE4n(n4e)`

`computeE4n` returns the $|\mathcal{N}_\ell| \times |\mathcal{N}_\ell|$ sparse matrix `e4n`. In this matrix the entry (j,k) is the number of the element which contains the nodes j and k as vertices in counterclockwise order, or zero if there is no such element,

$$\texttt{e4n(j,k)} := \begin{cases} m & \text{if } z_j, z_k \in \mathcal{N}(T_m),\ z_j,\ z_k \text{ ordered counterclockwise w.r.t. } T_m, \\ 0 & \text{otherwise.} \end{cases}$$

In `computeE4n` a matrix of all sides (i.e., all occuring pairs of nodes) and a vector of the corresponding element numbers is created. With the help of this matrix and vector, a sparse matrix is generated which contains all elements (i.e., element numbers) of node k in the k-th row.

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
nrElems = size(n4e,1);
N = max(max(n4e));
elemNumbers = [1:nrElems 1:nrElems 1:nrElems];
e4n = sparse(allSides(:,1),allSides(:,2),elemNumbers,N,N);
```

The application of `computeE4n` to `n4e` from the example in Figure 1.3.3 on page 11 results in a 15×15 matrix

$$\texttt{e4n} = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 3 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \cdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 17 & 0 \end{pmatrix}.$$

For example, the node patch of z_2 consists of the elements T_2, T_3 and T_4.

- Nodes for sides
  `n4s = computeN4s(n4e)`

`computeN4s` returns the $|\mathcal{S}_\ell| \times 2$ matrix `n4s`. For each $z_j, z_k \in \mathcal{N}_\ell$ with $\text{conv}\,(z_j, z_k) \in \mathcal{S}_\ell$, there is a row `[j k]` or `[k j]` in `n4s`. For boundary sides `n4s` contains a row `[j k]` such that z_k and z_j are in counterclockwise order. Whenever a side numbering is used, the one generated by `n4s` is meant.

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];

[b,ind] = unique(sort(allSides,2),'rows','first');
n4s = allSides(sort(ind),:);
```

`n4s` is a 32×2 matrix for the triangulation shown in Figure 1.3.3 on page 11.

$$\texttt{n4s} = \begin{pmatrix} 1 & 2 & \dots & 10 & 9 \\ 7 & 5 & \dots & 7 & 8 \end{pmatrix}^T.$$

E.g., S_1 connects z_1 and z_7.

- Sides for nodes
  `s4n = computeS4n(n4e)`

`computeS4n(n4e)` returns the symmetric $|\mathcal{N}_\ell| \times |\mathcal{N}_\ell|$ sparse matrix `s4n`. Define $S_m \in \mathcal{S}_\ell$ as $S_m = \mathrm{conv}(z_j, z_k)$, for $z_j, z_k \in \mathcal{N}_\ell$, then

$$\texttt{s4n(j,k)} = \texttt{s4n(k,j)} := \begin{cases} m & \text{if } \mathrm{conv}\,(z_j, z_k) = S_m \in \mathcal{S}_\ell, \\ 0 & \text{otherwise.} \end{cases}$$

Note that each side is encountered only once.

```
S = size(n4s,1);
N = max(max(n4e));
s4n = sparse(n4s(:,1),n4s(:,2),1:S,N,N);
```

Having incorporated each side in only one direction, the next step completes `s4n`.

```
s4n = s4n + s4n';
```

For the triangulation from Figure 1.3.3 `s4n` reads

$$\texttt{s4n} = \begin{pmatrix} 0 & 16 & 0 & 0 & 0 & 0 & 1 & 28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 16 & 0 & 18 & 0 & 0 & 2 & 29 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \cdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 26 & 13 & 0 \end{pmatrix}.$$

For example, S_{18} connects z_2 and z_3 but there is no side connecting z_1 and z_3.

- Sides for elements
  `s4e = computeS4e(n4e)`

`computeS4e` returns the $|\mathcal{T}_\ell| \times 3$ matrix `s4e` containing the side numbers for each element. The sides are ordered counterclockwise beginning with the side between the first and the second node of `n4e`. Let `n4e(j,:) = [a b c]`,

$$\mathrm{conv}\,(z_a, z_b) = S_r, \quad \mathrm{conv}\,(z_b, z_c) = S_s, \quad \mathrm{conv}\,(z_c, z_a) = S_t, \quad \{S_r, S_s, S_t\} = \mathcal{S}(T_j),$$

then `s4e(j,:) = [r s t]`. The matrix `s4e` is computed in a similar way as `n4s`.

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
[b,ind,back] = unique(sort(allSides,2),'rows','first');
[n4sInd, sortInd] = sort(ind);
sideNr(sortInd) = 1:length(ind);
s4e = reshape(sideNr(back),size(n4e));
```

Applied to the triangulation from Figure 1.3.3, `computeS4e` returns an 18×3 matrix

$$\texttt{s4e} = \begin{pmatrix} 1 & 1 & \dots & 13 & 14 \\ 15 & 16 & \dots & 27 & 10 \\ 28 & 29 & \dots & 26 & 27 \end{pmatrix}^T.$$

E.g., $S(T_2) = \{S_1, S_{16}, S_{29}\}$.

- Elements for sides
  `e4s = computeE4s(n4e)`

`computeE4s` returns the $|\mathcal{S}_\ell| \times 2$ matrix `e4s` containing the numbers of those elements in row j which share the j-th side. If the side is part of the boundary, the second entry is zero.

$$\texttt{e4s(j,:)} := \begin{cases} \texttt{[n k]} & \text{if } T_n \cap T_k = S_j \in \mathcal{S}_{\ell,\Omega}, \\ \texttt{[n 0]} & \text{if } S_j \in \mathcal{S}(T_n) \text{ and } S_j \in \mathcal{S}_{\ell,D} \cup \mathcal{S}_{\ell,N}. \end{cases}$$

Similarly to `s4e`, `e4s` is computed by indexing from `n4s`.

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
[b,ind,back] = unique(sort(allSides,2),'rows','first');
n4sInd = sort(ind);

nrElems = size(n4e,1);
elemNumbers = [1:nrElems 1:nrElems 1:nrElems];
e4s(:,1) = elemNumbers(n4sInd);
allElem4s(ind) = accumarray(back,elemNumbers);
e4s(:,2) = allElem4s(n4sInd)'-e4s(:,1);
```

Considering the triangulation from Figure 1.3.3 on page 11, `e4s` is a 32×2 matrix

$$\texttt{e4s} = \begin{pmatrix} 2 & 4 & \dots & 12 & 13 \\ 1 & 3 & \dots & 11 & 0 \end{pmatrix}^T.$$

For example, the boundary side S_{32} belongs only to the element T_{13}, while S_1 is shared by T_1 and T_2.

Element-Specific Data

The following functions provide element-specific information like areas of elements. This data depends on the structure of the used mesh `n4e`, as well as on the coordinates of each node `c4n`. Any set of elements given as a matrix where each row represents one element is also admissible as input instead of `n4e`. For the sake of simplicity, in all examples `n4e` is used as input. For any other set of elements as input, the dimensions of the output will change accordingly.

- Area for elements
  `area4e = computeArea4e(c4n,n4e)`

`computeArea4e` returns `area4e`, a $|\mathcal{T}_\ell|$-dimensional column vector containing the areas of the elements of $\mathcal{T}_\ell$. The area of a triangle $T = \text{conv}\{a, b, c\}$ is given by

$$|T| := \frac{1}{2} \left(a_x \left(b_y - c_y \right) + b_x \left(c_y - a_y \right) + c_x \left(a_y - b_y \right) \right).$$

This formula is implemented in a vectorised manner for all elements.

```
area4e = ( x1.*(y2 - y3) + x2.*(y3 - y1) + x3.*(y1 - y2) )/2;
```

- Area for node patches
  `area4n = computeArea4n(c4n,n4e)`

`computeArea4n` returns the $|\mathcal{N}_\ell|$-dimensional column vector `area4n` containing the areas of the node patches for all nodes (cf. Definition 1.3.3 on page 9). The vector `area4n` is computed using `area4e` by accumulation over `n4e`.

```
area4e = area4e * ones(1,3);
nrNodes = size(c4n,1);
area4n = accumarray(n4e(:),area4e(:),[nrNodes 1]);
```

- Midpoints for elements
  `mid4e = computeMid4e(c4n, n4e)`

`computeMid4e` returns the $|\mathcal{T}_\ell| \times 2$ matrix `mid4e` containing the two coordinates of the midpoints for each element. The midpoint of a triangle $T = \operatorname{conv}\{a, b, c\}$ is given by

$$\operatorname{mid}(T) := \frac{1}{3}(a + b + c).$$

This formula is evaluated simultaneously for all elements.

```
mid4e = ( c4n(n4e(:,1),:) + c4n(n4e(:,2),:) + c4n(n4e(:,3),:) ) / 3;
```

- Tangents for elements
  `tangent4e = computeTangent4e(c4n,n4e)`

`computeTangent4e` returns the $3 \times 2 \times |\mathcal{T}_\ell|$ matrix `tangent4e`. Each 3×2 submatrix contains the coordinates of the tangent vectors oriented counterclockwise w.r.t. corresponding element. Let $S = \operatorname{conv}\{a, b\} \in \mathcal{S}_\ell$, then the unit tangent vector τ_S is given by

$$\tau_S := \frac{b - a}{|S|}. \tag{1.9.1}$$

The corresponding Matlab implementation reads

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
c4start = c4n(allSides(:,1),:);
c4end = c4n(allSides(:,2),:);
lengths = sqrt(sum((c4end-c4start).^2,2));
tangents = (c4end - c4start)./[lengths lengths];
tangent4e(1,:,:) = tangents(1:size(n4e,1),:)';
tangent4e(2,:,:) = tangents(size(n4e,1)+1:2*size(n4e,1),:)';
tangent4e(3,:,:) = tangents(2*size(n4e,1)+1:3*size(n4e,1),:)';
```

- Normals for elements
  `normal4e = computeNormal4e(c4n,n4e)`

`computeNormal4e` returns the $3 \times 2 \times |\mathcal{T}_\ell|$ matrix `normal4e` containing the outer unit normal vectors for each side of each element. Given the unit tangent vector τ_S of a side $S \in \mathcal{S}_\ell$, the outer unit normal vector ν_S is given by

$$\nu_S := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tau_S. \tag{1.9.2}$$

The corresponding Matlab implementation reads

```
allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
c4start = c4n(allSides(:,1),:);
c4end = c4n(allSides(:,2),:);
lengths = sqrt(sum((c4end-c4start).^2,2));
tangents = (c4end - c4start)./[lengths lengths];
normals = [tangents(:,2), -tangents(:,1)];
normal4e(1,:,:) = normals(1:size(n4e,1),:)';
normal4e(2,:,:) = normals(size(n4e,1)+1:2*size(n4e,1),:)';
normal4e(3,:,:) = normals(2*size(n4e,1)+1:3*size(n4e,1),:)';
```

Side-Specific Data

The following functions provide side-specific information for a triangulation, like the lengths of its sides. This data depends on the sides within the used mesh (i.e., `n4s` as generated by the function `computeN4s`) and on the coordinates of each node (i.e., `c4n`). Any set of sides given as a matrix similar to `n4s` is also admissible as input instead of `n4s`. For the sake of simplicity, in all examples `n4s` is used as input. For any other set of sides in the input, the dimensions of the output will change accordingly.

- Length for sides
  `length4s = computeLength4s(c4n,n4s)`

`computeLength4s` returns the $|\mathcal{S}_\ell|$-dimensional column vector `length4s`. For each given side $S = \mathrm{conv}\{a, b\} \in \mathcal{S}_\ell$, the length is computed by

$$|S| := \sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}.$$

The corresponding Matlab implementation reads

```
length4s = sqrt( sum( (c4n(n4s(:,2),:) - c4n(n4s(:,1),:)).^2, 2) );
```

- Midpoints for sides
  `mid4s = computeMid4s(c4n, n4s)`

`computeMid4s` returns the $|\mathcal{S}_\ell| \times 2$ matrix `mid4s`. For each given side $S = \mathrm{conv}\{a, b\}$ the midpoint is computed by

$$\mathrm{mid}(S) := \frac{1}{2}(a + b).$$

This formula is implemented in `computeMid4s`.

```
mid4s = 0.5 * ( c4n(n4s(:,1),:) + c4n(n4s(:,2),:) );
```

- Tangents for sides
  `tangent4s = computeTangent4s(c4n,n4s)`

`computeTangent4s` returns the $|\mathcal{S}_\ell| \times 2$ matrix `tangent4s`. For each given side $S = \operatorname{conv}\{a, b\}$ of some triangle T_+ induced by `computeE4s`, the unit tangent vector τ_S along S w.r.t. T_+ is computed by (1.9.1), cf. Figure 1.8.1 on page 33. For boundary sides the orientation of τ_S is determined by the orientation of its element $T(S)$, cf. Figure 1.3.2 on page 10. The corresponding Matlab implementation reads

```
tangent4s = (c4end-c4start)./[length4s length4s];
```

- Normals for sides
  `normal4s = computeNormal4s(c4n,n4s)`

`computeNormal4s` returns the $|\mathcal{S}_\ell| \times 2$ matrix `normal4s`. For each given side $S = \operatorname{conv}\{a, b\}$ of some triangle T_+, the unit normal vector ν_S pointing outward w.r.t. T_+ is computed by (1.9.2), cf. Figure 1.8.1 on page 33. The enumeration introduced in `computeE4s` induces the triangle T_+ for each side. For boundary sides, ν_S is the outer unit normal given by its element $T(S)$.

```
normal4s = [tangent4s(:,2), -tangent4s(:,1)];
```

1.9.2 Plot Functions

The AFEM package includes functions for plotting finite element solutions, triangulations and convergence graphs as well as estimated errors on sides or elements of the triangulation.

General Plot Functions

The following functions are usable for any type of element. All example plots were generated by solving the Poisson model problem (1.2.2) on an L-shaped domain.

- Plot Triangulation
  `plotTriangulation(c4n,n4e)`

`plotTriangulation` plots the triangular grid defined by `c4n` and `n4e`. A typical output of `plotTriangulation` is shown in Figure 1.9.2.

- Plot Convergence Graph
  `plotConvergence(nrDoF4lvl, error4lvl, varargin)`

`plotConvergence` creates a plot of error values (given by `error4lvl`) over the number of degrees of freedom (given by `nrDoF4lvl`) in double logarithmic scaling. The optional `titel` and the convergence rate are added to the legend. Such plots are useful to observe the convergence behaviour of the discrete solution for a specific finite element and specific error norm. A typical plot is shown in Figure 1.9.3.

- Plot P_0 Function on Sides
  `plotP04s(c4n, n4e, x, OPTtitle)`

`plotP04s` draws a sidewise P_0 function on the sides of the triangulation (`c4n`, `n4e`) where each entry of `x` represents the value on the corresponding side. On each side of the triangulation a vertical patch is drawn on which the height and colour represent the value of the function. The optional input argument `OPTtitle` sets the title of the figure. The default value is empty. A typical plot is shown in Figure 1.9.5.

- Plot P_0 Function on Elements
  `plotP04e(c4n, n4e, x, OPTtitle)`

`plotP04e` draws an elementwise P_0 function on the triangulation (`c4n`, `n4e`) given by `x` where each entry of `x` represents the value on the corresponding element. On each element of the triangulation a horizontal patch is drawn on which the height and colour represent the corresponding function value. The optional input argument `OPTtitle` sets the title of the figure. The default value is empty. A typical plot is shown in Figure 1.9.6.

Element-Specific Plot Functions

The following functions plot solutions for specific types of elements (P_1, CR and RT_0).

- Plot P_1 Function
  `plotP1(c4n,n4e,x,OPTtitle)`

`plotP1` draws a P_1 function given by the coefficients `x` of the standard nodal basis functions on a triangulation (`c4n`, `n4e`). The optional input argument `OPTtitle` sets the title of the figure. The default value is empty. Figure 1.9.7 shows a P_1-solution to the Poisson model problem on an L-shaped domain. Matlab provides the function `trisurf` which draws a piecewise P_1 function on a triangular mesh, which is used in `plotP1` to plot the solution. If the number of elements in the triangulation is less or equal 2000, the mesh is plotted as well.

```
9  if(size(n4e,1)>2000)
10    trisurf(n4e,c4n(:,1),c4n(:,2),x,'EdgeColor','none');
   else
      trisurf(n4e,c4n(:,1),c4n(:,2),x);
   end
```

Figure 1.9.2: A refined grid with 189 nodes on an L-shaped domain plotted by `plotTriangulation`.

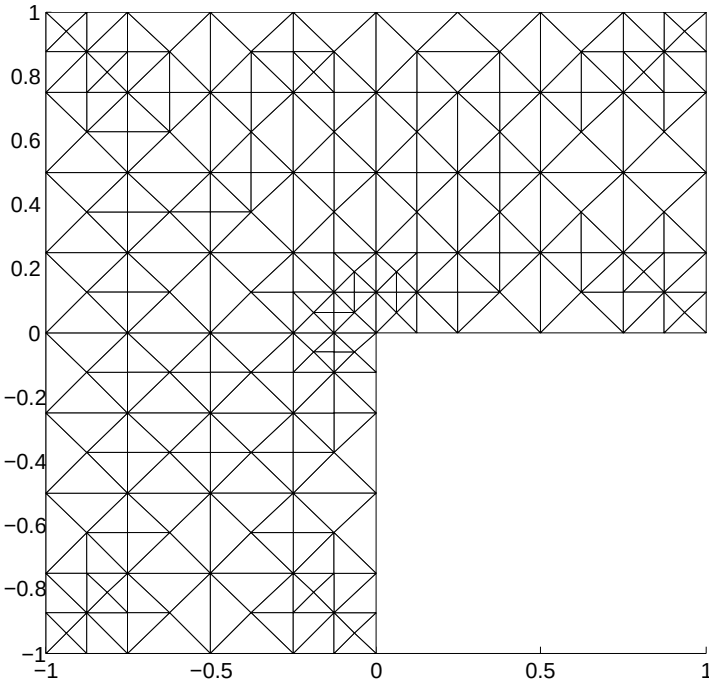
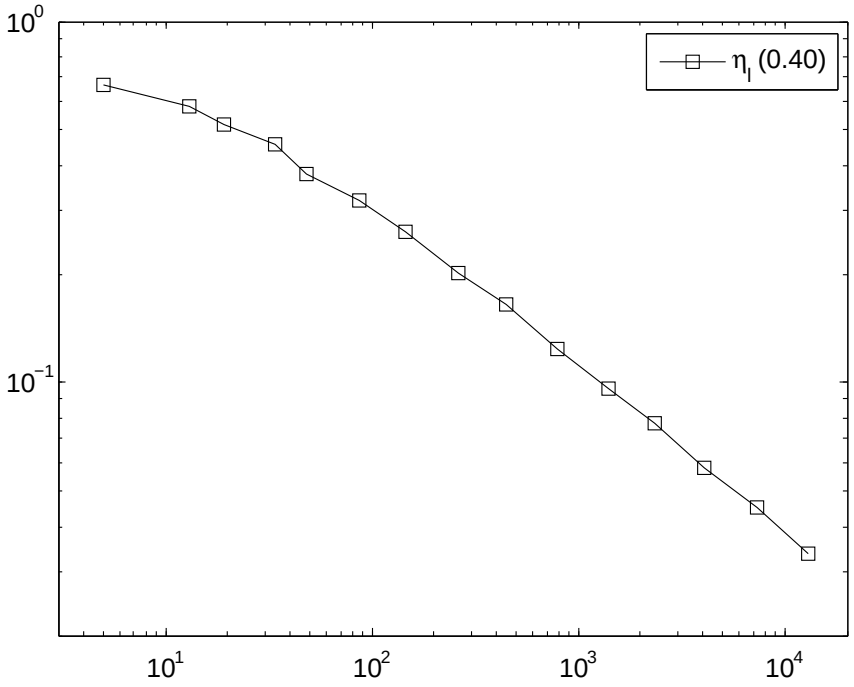


Figure 1.9.3: Estimated error over number of degrees of freedom plotted using the function `plotConvergence`.



```
15 set(handle,'Name','');
   if nargin == 4
           title(OPTtitle);
       else
           title('');
20 end
   drawnow;
```

- Plot *CR* Function
  `plotCR(c4n,n4e,x,OPTtitle)`

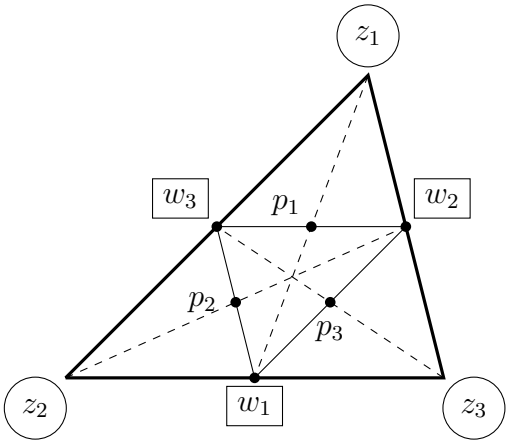
Based on `c4n`, `n4e` and the coefficients `x` for the basis functions, `plotCR` draws a Crouzeix-Raviart function. The optional input argument `OPTtitle` sets the title of the figure. The default value is empty. Figure 1.9.8 shows a Crouzeix-Raviart solution of the Poisson model problem on an L-shaped domain. In order to plot a discrete Crouzeix-Raviart solution, the values of the degrees of freedom w_1, w_2, w_3 (at the midpoints of the sides) need to be translated into values at the vertices z_1, z_2, z_3. Each element can be decomposed into four triangles such that each outer triangle is point symmetric to the inner one as depicted in Figure 1.9.4. Thus the values for the vertices can be computed as follows:

$$z_j = p_j + (p_j - w_j) = 2p_j - w_j, \quad j \in \{1, 2, 3\}.$$

Using $p_1 = (w_2 + w_3)/2$ and the analogous equations for p_2 and p_3, this yields:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}.$$

Figure 1.9.4: Relationship of degrees of freedom and nodes in the Crouzeix-Raviart element used in `plotCR`.



- Plot RT_0 Function
  `plotRT0(c4n,n4e,u,p,OPTtitle4u,OPTtitle4p)`

`plotRT0` visualises the discrete function `u` with its flux `p` in two different figures. The solution `u` is drawn by `plotP04e`. The function `quiver2.m` is used to plot the flux `p` (cf. [1]). The input arguments are the triangulation given by `c4n`, `n4e`, the solution `u` and its flux `p` returned by `solveRT0Poisson` and two optional parameters `OPTtitle4u` and `OPTtitle4p`, which define the title of the figures for `u` and `p`, respectively. The default value for both optional parameters is empty.

Figure 1.9.5: Estimated error on sides of the triangulation plotted using the function `plotP04s`.

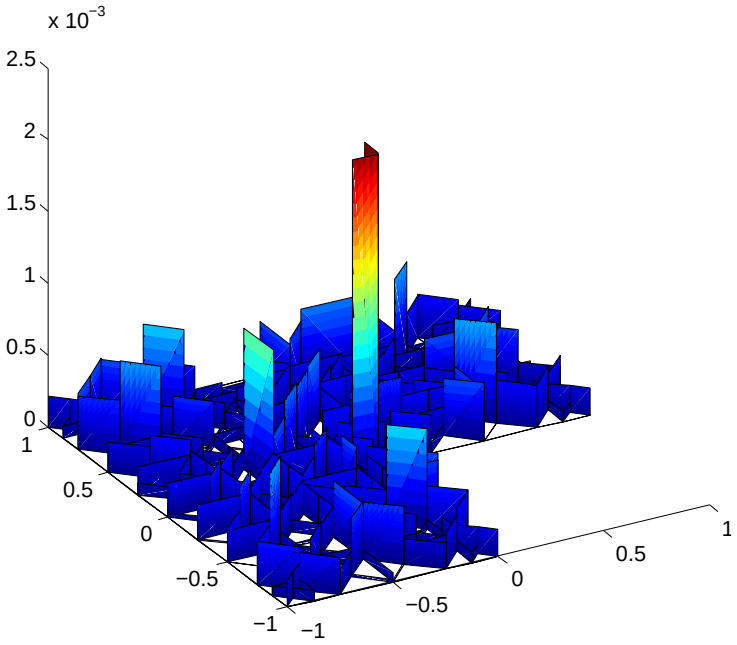


Figure 1.9.6: Estimated error on elements of the triangulation plotted using the function `plotP04e`.

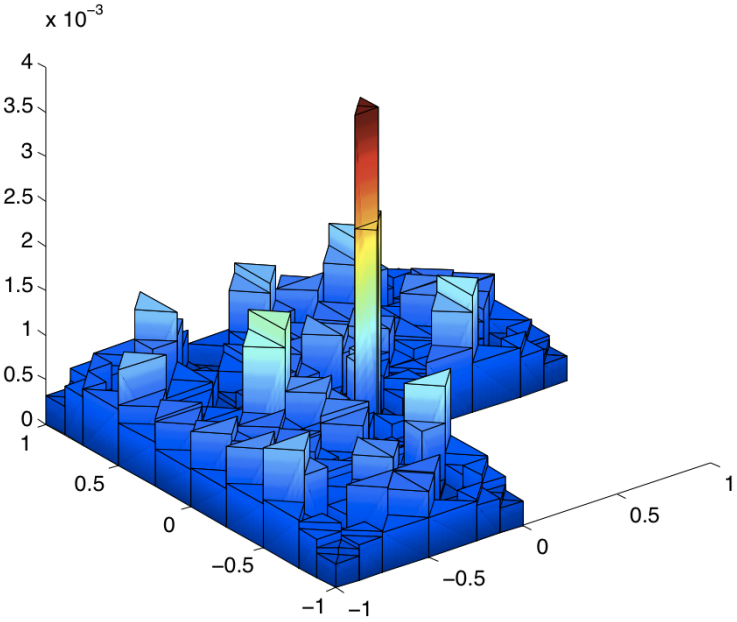


Figure 1.9.7: A P_1 solution of the Poisson model problem with 189 nodes on an L-shaped domain plotted by `plotP1`.

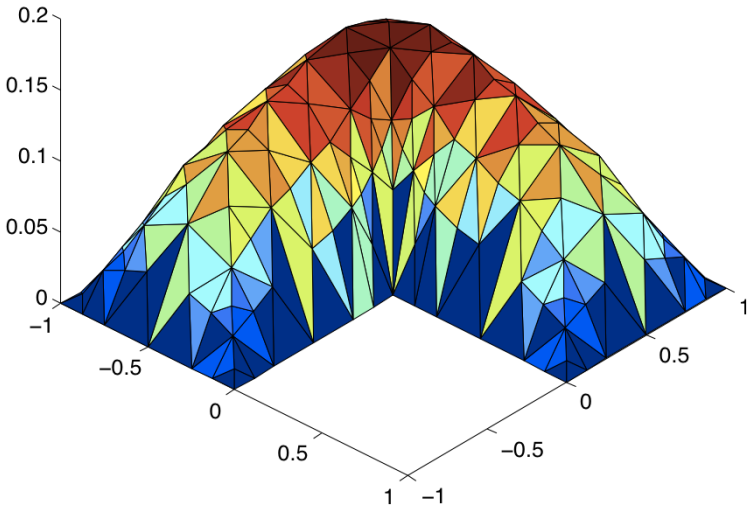
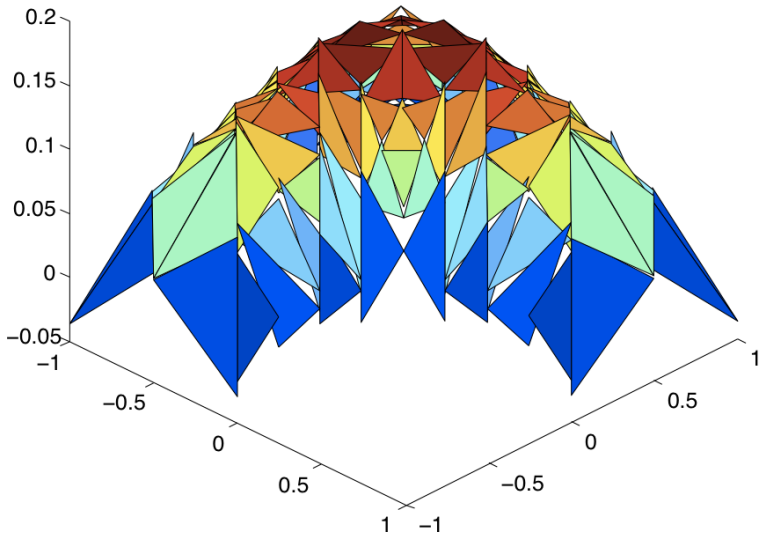


Figure 1.9.8: A Crouzeix-Raviart solution with 220 sides on an L-shaped domain plotted by `plotCR`.



2 Applications

2.1 Example Programs for the Poisson Problem Using the AFEM Package

This section is devoted to various implementations which demonstrate the application of AFEM for solving the Poisson model problem. A detailed introduction to the definition of the problem input data, as well as an initial triangulation of the domain Ω can be found in Sections 1.2 and 1.3 whereas the use of the special functions of the AFEM cycle SOLVE, ESTIMATE, MARK and REFINE is given in Sections 1.4 - 1.7. In addition to the P_1, CR and RT_0 implementations of the AFEM cycle for the Poisson problem there are another three files `afemP1PoissonSquareExact` with $u(x,y) = x(1-x)y(1-y)$, `afemCRPoissonLShapeExact` with $u(r,\varphi) = r^{(2/3)}\sin(\frac{2}{3}\varphi)$ and `afemRT0PoissonSlitExact` with $u(r,\varphi) = r^{(1/4)}\sin(\frac{1}{4}\varphi)$ to compare the convergence behaviour of the estimator and the exact L_2 error for a known solution.

2.1.1 AFEM with P_1 Finite Elements

`afemP1PoissonSquareExact.m` realises the AFEM cycle for conform P_1 finite elements to solve the Poisson model problem and to compare the discrete P_1 and the exact solutions. The right-hand side `f`, Dirichlet boundary data `u4Db`, Neumann boundary data `g` and the exact data `uExact` and `gradExact` are given as follows.

```
function val = f(x)
    val = 2*x(:,1) - 2*x(:,1).^2 + 2*x(:,2) - 2*x(:,2).^2;
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    x1 = x(:,1);
    x2 = x(:,2);
    for i=1:(size(x,1))
        if x1(i)==0
            N = [-1;0];
        elseif x2(i)==0
            N = [0;-1];
        elseif x1(i)==1
            N = [1;0];
        elseif x2(i)==1
```

```
            N = [0;1];
        end
        val(i,:) = (x2(i) - x2(i)^2 - 2*x1(i)*x2(i) +...
            2*x1(i)*x2(i)^2)*N(1,1) + (x1(i) - x1(i)^2 - 2*x1(i)*x2(i) +...
            2*x2(i)*x1(i)^2)*N(2,1);
    end
end

function val = uExact(x)
    x1=x(:,1);
    x2=x(:,2);
    val = x1.*(1-x2).*x2.*(1-x1);
end

function val = gradExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = [x2 - x2.^2 - 2*x1.*x2 + 2*x1.*x2.^2,...
        x1 - x1.^2 - 2*x1.*x2 + 2*x2.*x1.^2];
end
```

To access all functions of AFEM the root directory and all its subdirectories have to be added to the path. The geometrical data of the unit square with Neumann boundaries `SquareNb` is loaded using `loadGeometry` and uniformly refined once. The AFEM loop continues until the number of degrees of freedom has reached at least 1000. Fields are initialised for keeping track of the error indicator, the exact L^2 error and the energy error for each computed level.

```
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('SquareNb',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);
    error4nrDoF = sparse(1,1);
    energy4nrDoF = sparse(1,1);
```

The AFEM cycle itself is implemented as a loop which is left with the `break` command as soon as a given number of degrees of freedom is reached.

```
    while( true )
        % SOLVE
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        %Exact error
        error4e = error4eP1L2(c4n,n4e,@uExact,x);
        error4nrDoF(nrDoF) = sqrt(sum(error4e));
        %Energy error
        energy4e = error4eP1Energy(c4n,n4e,@gradExact,x);
        energy4nrDoF(nrDoF) = sqrt(sum(energy4e));
        % ESTIMATE
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
            eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
```

```
35        % MARK
          n4sMarked = markBulk(n4s,eta4s);
          % REFINE
          [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
      end
```

After this, the final mesh, the discrete and the exact solutions on that mesh and a convergence graph are drawn using the respective functions. Figure 2.1.2 shows an example output for the unit square of `afemP1PoissonSquareExact`.

```
47    figure;
      plotTriangulation(c4n,n4e);
      figure;
50    plotP1(c4n,n4e,x,{'P1 Solution' [num2str(length(x)) ' nodes']});
      nrDoF4lvl = find(eta4nrDoF);
      error4lvl = error4nrDoF(nrDoF4lvl);
      eta4lvl = eta4nrDoF(nrDoF4lvl);
      energy4lvl = energy4nrDoF(nrDoF4lvl);
55    figure;
      plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
      hold all;
      plotConvergence(nrDoF4lvl,error4lvl,'||u - u_l||_{L2}');
      plotConvergence(nrDoF4lvl,energy4lvl,'||\nablau - \nablau_l||_{L2}');
```

Figure 2.1.1: Convergence behaviour of errors plotted by `afemP1PoissonSquareExact` with `minNrDof` =10 000 on the unit square and initial mesh `SquareNb`

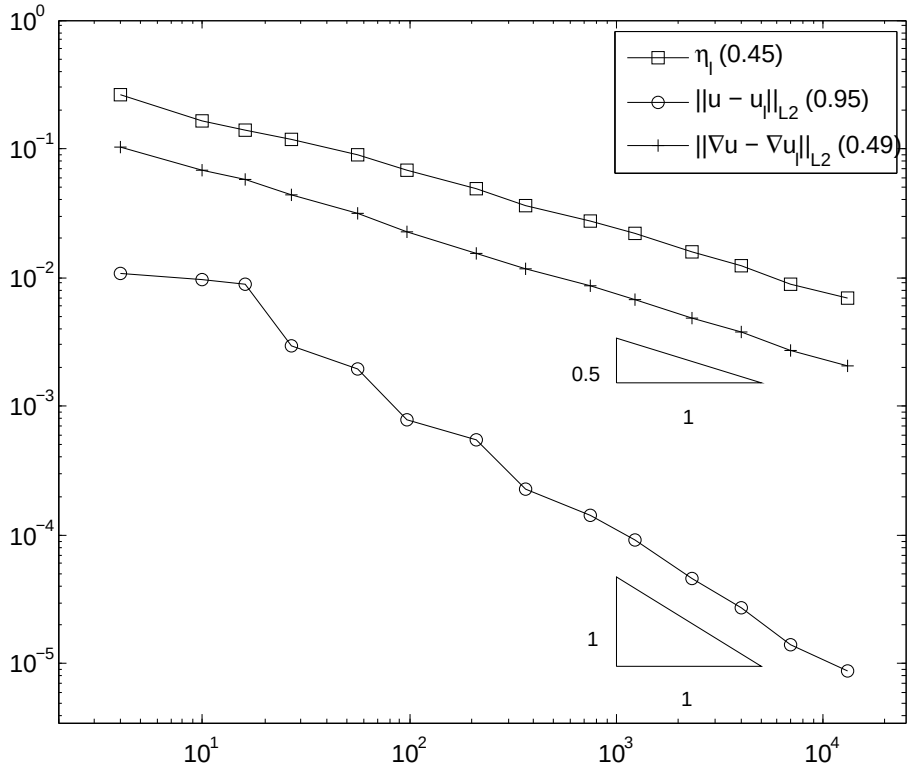
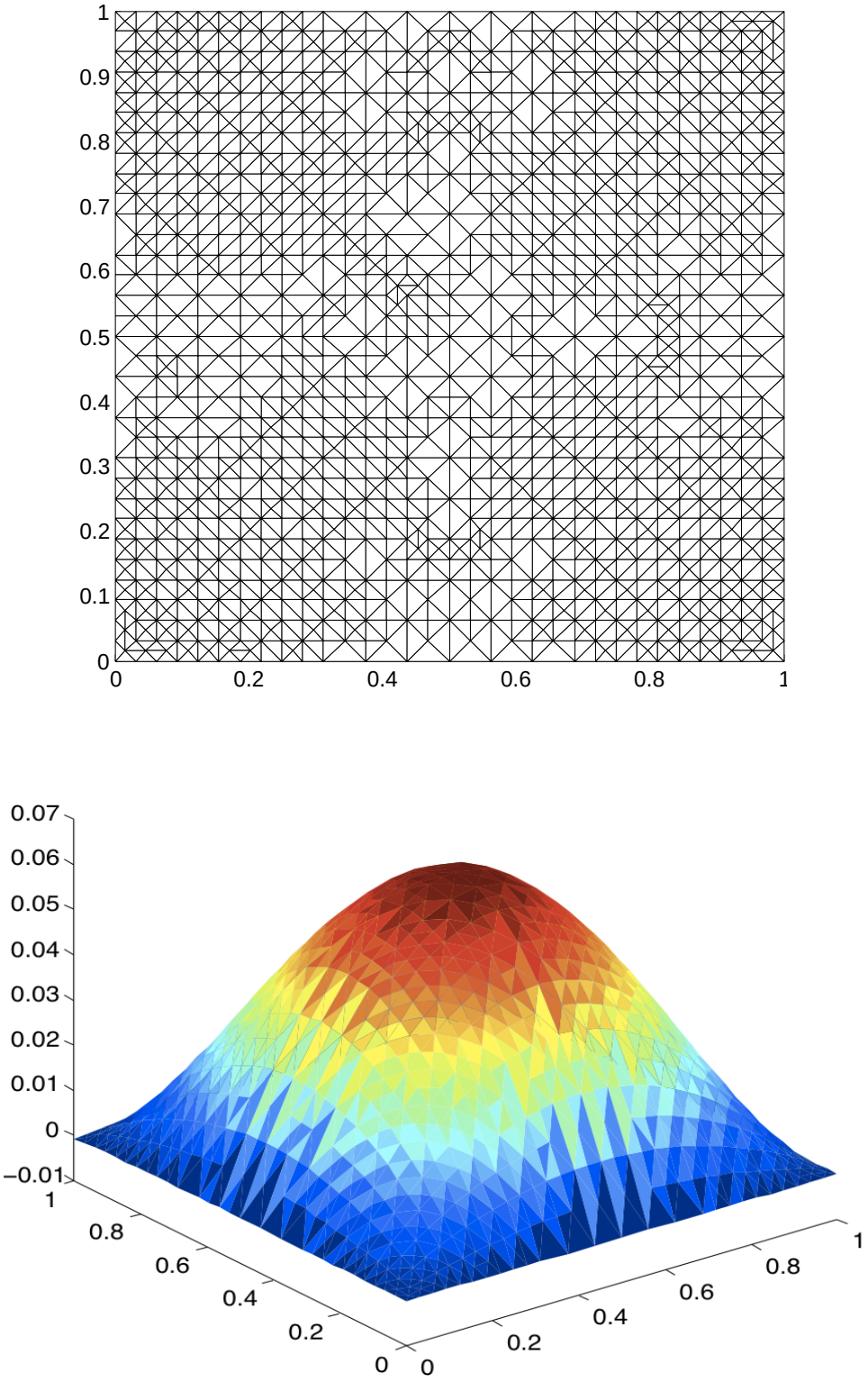


Figure 2.1.2: Mesh and solution with 1293 nodes plotted by `afemP1PoissonSquareExact` with `minNrDof = 1 000` on the unit square and initial mesh `SquareNb`



2.1.2 AFEM with *CR* Finite Elements

A simple program implementing the AFEM cycle for the Poisson model problem with non-conforming Crouzeix-Raviart elements can be found in `afemCRPoissonLShapeExact.m`. The program solves the Poisson problem on a given L–shape geometry and compares the discrete *CR* with the given exact solution. The right-hand side `f`, the Dirichlet boundary data `u4Db`, the Neumann boundary data `g` and the exact data `uExact` and `gradExact` are given as in the conforming case in the previous Subsection 2.1.1. To solve the model problem, the AFEM cycle is executed until a given number of degrees of freedom is reached.

```
24    while( true )
25        % SOLVE
          [x,nrDoF] = solveCRPoisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
          nrDoF4lvl(end+1) = nrDoF;
          %Exact error
          error4lvl(end+1) = sqrt(sum(error4eCRL2(c4n, n4e, @uExact, x)));
30        %Energy error
          energy4lvl(end+1) = ...
             sqrt(sum(error4eCREnergy(c4n, n4e, @gradExact, x)));
          % ESTIMATE
          [eta4s,n4s] = estimateCREtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
35        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
          disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
              '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
          if nrDoF >= minNrDoF, break, end;
          % MARK
40        n4sMarked = markBulk(n4s,eta4s);
          % REFINE
          [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
      end
```

Afterwards, the discrete *CR* solution is plotted using `plotCR` introduced in Section 1.9.2. The corresponding mesh and a convergence graph are displayed by means of the respective functions from the AFEM package as in the previous implementation 2.1.1.

Figure 2.1.3: Convergence behaviour of errors plotted by `afemCRPoissonLShapeExact` with `minNrDof =` 10 000 on the L-shape domain and initial mesh `LShapeNb`

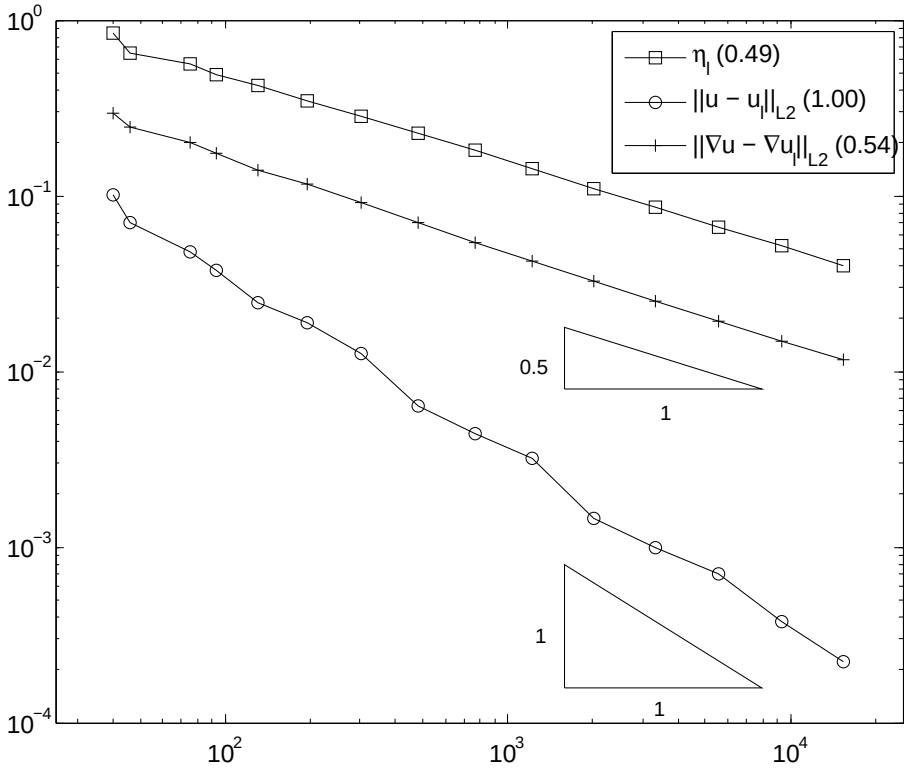
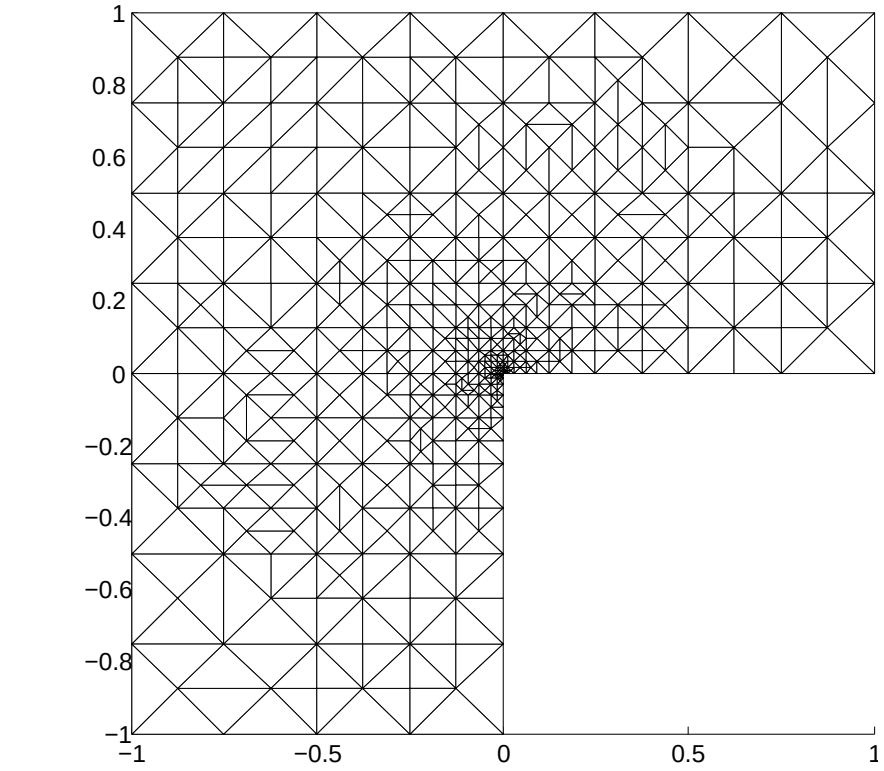
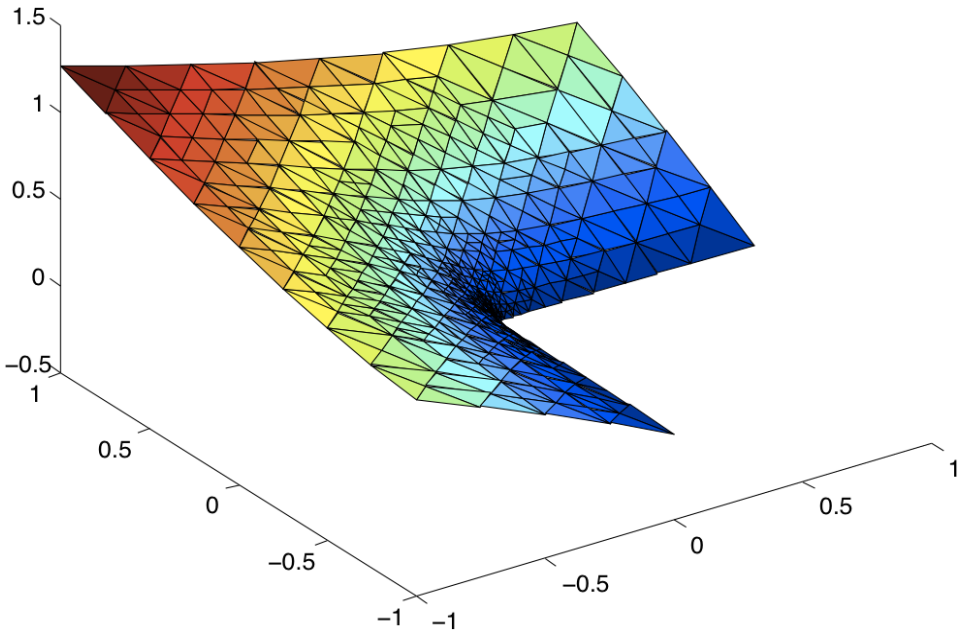


Figure 2.1.4: Mesh with 435 nodes and solution with 1252 sides plotted by `afemCRPoissonLShapeExact` with `minNrDof = 1 000` on the L-shape domain and intital mesh `LShapeNb`





2.1.3 AFEM with RT_0 Finite Elements

The example program `afemRT0PoissonSlitExact.m` uses the RT_0 elements introduced in Section 1.4.3 as well as the error indicator from Section 1.5.5 to solve the Poisson problem on a slit geometry and to compare the discrete RT_0 with the exact solution. The right-hand side `f`, the Dirichlet boundary data `u4Db`, the Neumann boundary data `g` and the exact data `uExact` and `gradExact` are given as in the conforming case in Subsection 2.1.1. The AFEM cycle is executed until the given number of degrees of freedom is reached.

```
    while( true )
        % SOLVE
        [p,u,nrDoF] = solveRT0Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        %Exact error
        error4e = error4eRT0L2(c4n, n4e, @uExact, u);
        error4lvl(end+1) = sqrt(sum(error4e));
        %Energy error
        energy4e = error4eRT0Energy(c4n, n4e, @gradExact, p);
        energy4lvl(end+1) = sqrt(sum(energy4e));
        % ESTIMATE
        [eta4s,n4s] = estimateRT0EtaSides(@f,@g,@u4Db,p,u,c4n,n4e,n4sDb,n4sNb);
        eta4lvl(end+1) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
              '; estimator = ',num2str(eta4lvl(end))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end
```

Afterwards the mesh, the discrete RT_0 solution on that mesh and the convergence behaviour of errors are plotted. `plotP04e` and `quiver2` are used to illustrate the solutions for u and p respectively. Details of `quiver2` can be found in Section 3 `extern/quiver2.m`.

Figure 2.1.5: Convergence behaviour of errors plotted by `afemRT0PoissonSlitExact` with `minNrDof = 10 000` on the slit domain and initial mesh `SlitNb`

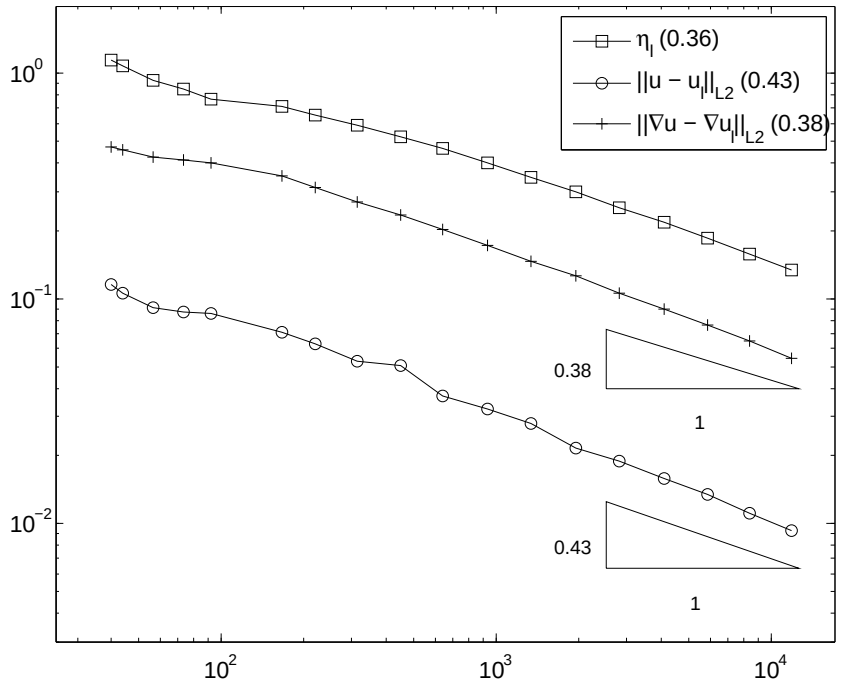


Figure 2.1.6: Mesh with 483 nodes plotted by `afemRT0PoissonSlitExact` with `minNrDof = 1 000` on the slit domain and initial mesh `SlitNb`

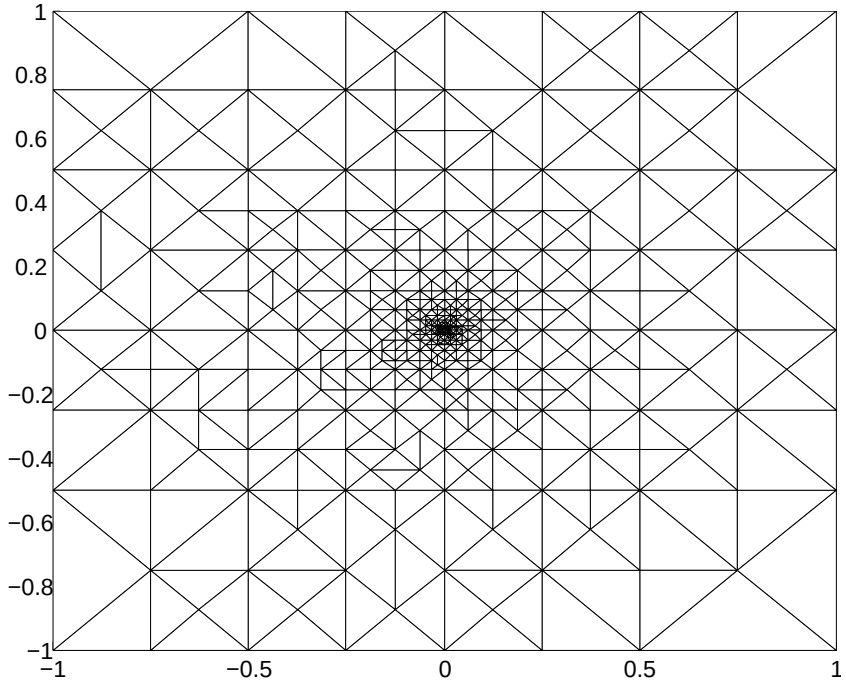
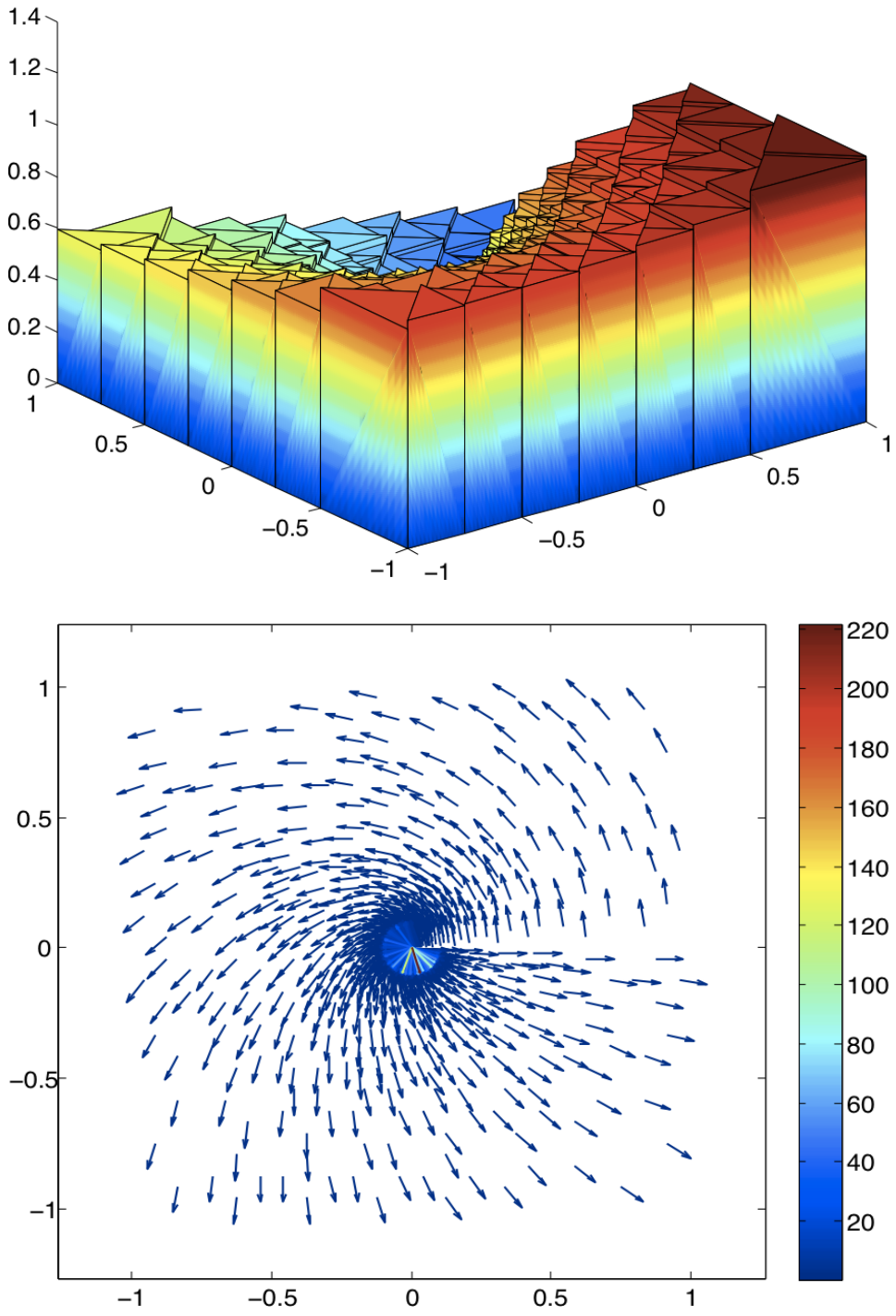


Figure 2.1.7: Solution for `u` and `p` with 1810 degrees of freedom plotted by `afemRT0PoissonSlitExact` with `minNrDof = 1 000` on the slit domain and initial mesh `SlitNb`



2.2 Example Programs for the Stokes' Equations Using the CR and P_0 Finite Elements

In this section, four AFEM implementations for the Stokes' equations using the CR element for the velocity u and the P_0 element for the pressure p will be presented. Each implementation is equipped with the two error estimators `estimateNCStokesEtaElements` and `estimateSigmaAveragingP1` (cf. sections 1.5.6 and 1.5.3). For the examples on the unit square ("colliding flow"), the L-shape and the slit domain exact solutions are known and will be compared to the approximated solutions by the exact errors. Thus, one can deduce hints on reliability and efficiency of the estimators. For the "backward facing step" example however an exact solution is unknown.

2.2.1 The Colliding Flow Example

The example program `afemCRP0StokesCollidingFlow.m` implements the AFEM cycle for the Stokes' equations on the unit square $(-1,1) \times (-1,1)$ with the right-hand side $f \equiv 0$ and the exact solution

$$u(x) = \begin{pmatrix} 20x_1x_2^4 - 4x_1^5 \\ 20x_1^4x_2 - 4x_2^5 \end{pmatrix} \quad \text{and} \quad p(x) = 120x_1^2x_2^2 - 20x_1^4 - 20x_2^4 - 16/3$$

for $x = (x_1, x_2) \in \overline{\Omega}$. The problem input data consisting of the right-hand side `f`, Dirichlet boundary data `u4Db`, its derivative `Du4Db1` and `Du4Db2`, Neumann boundary data `g` and the exact data `uExact1`, `uExact2`, `pExact`, `gradExact1`, `gradExact2`, `sigmaExact1` and `sigmaExact2` read

```
function val = uExact1(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = 20*x1.*x2.^4-4*x1.^5;
end

function val = uExact2(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = 20*x1.^4.*x2 - 4*x2.^5;
end

function val = f(x)
    val = zeros(size(x));
end

function val = u4Db(x)
    val = [uExact1(x),uExact2(x)];
end

function val = g(x)
    val = zeros(size(x));
end
```

```
function val = gradExact1(x)
    val(:,1) = 20*x(:,2).^4-20*x(:,1).^4;
    val(:,2) = 80*x(:,1).*x(:,2).^3;
end

function val = gradExact2(x)
    val (:,1) = 80*x(:,1).^3.*x(:,2);
    val(:,2) = 20*x(:,1).^4-20*x(:,2).^4;
end

function val = Du4Db1(x)
    val = gradExact1(x);
end

function val = Du4Db2(x)
    val = gradExact2(x);
end

function val = pExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = (120*x1.^2.*x2.^2-20*x1.^4-20*x2.^4-32/6);
end

function val = sigmaExact1(x)
    val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
end

function val = sigmaExact2(x)
    val = gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
end
```

By default the AFEM cycle will be executed until the number of degrees of freedom has reached at least 5000 and the bulk parameter is chosen as 0.5. As described in section 2.1.1 the AFEM root directory and its subdirectories are added to the path, the corresponding geometrical data is loaded and the arrays intended for the values of the error estimators and the exact errors are initialised. Afterwards the AFEM loop is executed the following way with simultaneous computation of the different errors and error estimators.

```
    while( true )
        %% SOLVE
        [u,p,~,~,nrDoF,gradU4e] = solveCRP0Stokes(@(x)f(x),@(x)u4Db(x),...
                                        @(x)g(x),c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        gradU4e1 = reshape(gradU4e(1,:,:),2,size(gradU4e,3))';
        gradU4e2 = reshape(gradU4e(2,:,:),2,size(gradU4e,3))';
        % Averaging estimated error
        averaging4lvl(end+1) = ...
               sqrt(sum(estimateSigmaAveragingP1(c4n,n4e,[gradU4e1 - ...
                   [p,zeros(size(p))],gradU4e2 - [zeros(size(p)),p]])));
```

```
    % L2 exact error
    error4lvl(end+1) = ...
           sqrt(sum(error4eCRL2(c4n,n4e,@(x)uExact1(x),u(:,1)))+...
                  sum(error4eCRL2(c4n,n4e,@(x)uExact2(x),u(:,2))));
    % Energy exact error
    energy4lvl(end+1) = ...
        sqrt(sum(error4eCREnergy(c4n,n4e,@(x)gradExact1(x),u(:,1)))+...
            sum(error4eCREnergy(c4n,n4e,@(x)gradExact2(x),u(:,2))));
    % Stress exact error
    stress4lvl(end+1) = ...
                  sqrt(sum(error4eStokesCRStress(c4n,n4e,1,...
                         @(x)sigmaExact1(x),u(:,1),p,gradU4e1))+...
                     sum(error4eStokesCRStress(c4n,n4e,2,...
                         @(x)sigmaExact2(x),u(:,2),p,gradU4e2)));
    %% ESTIMATE
    [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,...
                  @(x)f(x),@(x)Du4Db1(x),@(x)Du4Db2(x),u,gradU4e);
    eta4nrDoF(nrDoF) = sqrt(sum(eta4e));
    % print information on afem loop
    disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
        '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
    % break condition
    if nrDoF >= minNrDoF, break, end;
    %% MARK
    if theta==1
        n4sMarked = markUniform(n4e);
    else
        n4sMarked = markBulk(n4e,eta4e,theta);
    end
    %% REFINE
    [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
end
```

Finally the mesh, the discrete solution of the velocity u and the convergence history plot with the exact errors and the error estimators are drawn. In order to plot the velocity vector field one uses the `quiver2` function. Details of `quiver2` can be found in Section 3 `extern/quiver2.m`.

```
nrDoF4lvl = find(eta4nrDoF);
eta4lvl = eta4nrDoF(nrDoF4lvl);
% Mesh
figure;
plotTriangulation(c4n,n4e);
axis equal tight;
% Solution
mid4s = computeMid4s(c4n,n4s);
figure
quiver2(mid4s(:,1),mid4s(:,2),u(:,1),u(:,2),'n=',0.1,'w=',[1 1]);
% Convergence graphs
figure;
plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l adaptive');
hold all;
plotConvergence(nrDoF4lvl,error4lvl,'L2-Error adaptive');
```

```
    plotConvergence(nrDoF4lvl,energy4lvl,'Energy-Error adaptive');
    plotConvergence(nrDoF4lvl,stress4lvl,'||\sigma-\sigma_{CR}||_{L^2}');
85  plotConvergence(nrDoF4lvl,averaging4lvl,'Av-Error adaptive');
```

The following figures show an example output of `afemCRP0StokesCollidingFlow`.

Figure 2.2.1: Convergence behaviour of errors plotted by `afemCRP0StokesCollidingFlow` with `minNrDof = 5 000` on the big unit square domain and initial mesh `BigSquare`

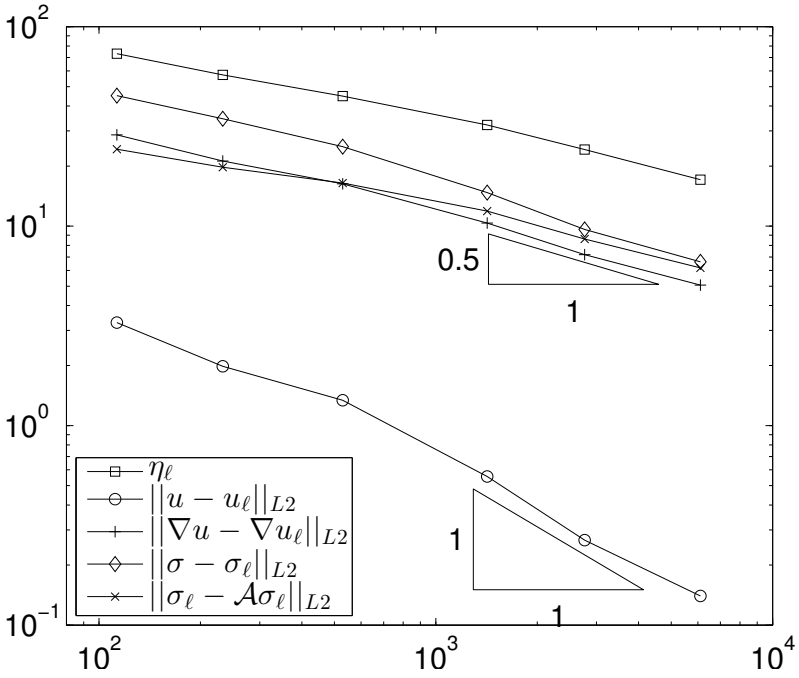


Figure 2.2.2: Mesh with 840 nodes plotted by `afemCRP0StokesCollidingFlow` with `minNrDof = 5 000` on the big unit square domain and initial mesh `BigSquare`

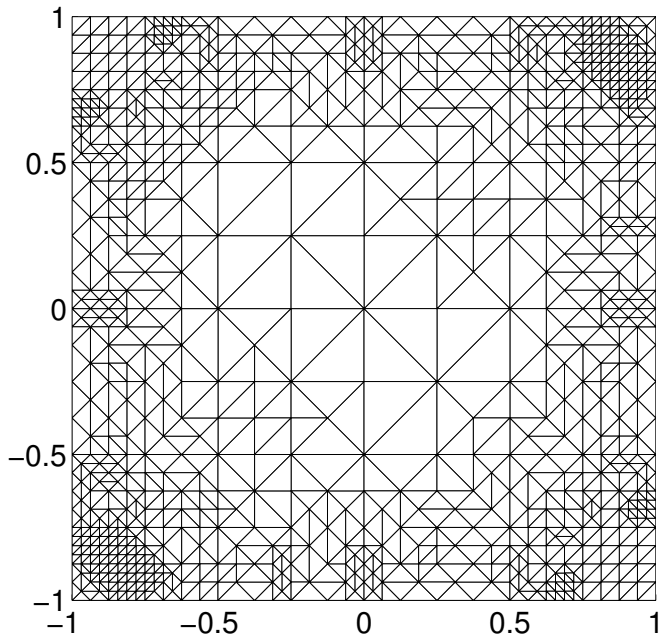
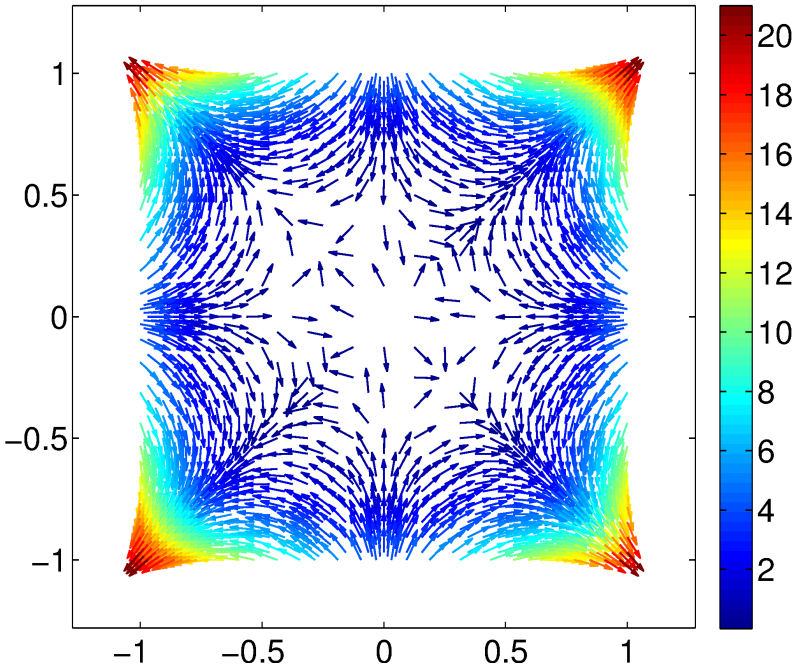


Figure 2.2.3: Solution for `u` with 6 103 degrees of freedom plotted by `afemCRP0StokesCollidingFlow` with `minNrDof = 5 000` on the big unit square domain and initial mesh `BigSquare`



2.2.2 The L-Shape Example

The example program `afemCRP0StokesLshapeExact.m` implements the AFEM cycle for the Stokes' equations on the L-shape domain $(-1,1)\times(-1,1)\setminus[0,1]\times[-1,0]$ with the right-hand side $f\equiv 0$. Let $(r,\theta)\in[0,\infty)\times[0,2\pi)$ be the polar coordinates of $x\in\overline{\Omega}$. Then the exact solution of the considered problem reads

$$u(x)=u(r,\theta)=r^{\alpha}w(\theta)\begin{pmatrix}(1+\alpha)\sin(\theta)+\cos(\theta)\\ -(1+\alpha)\cos(\theta)+\sin(\theta)\end{pmatrix}\quad\text{and}$$
$$p(x)=p(r,\theta)=-r^{\alpha-1}((1+\alpha)^2w'(\theta)+w'''(\theta))/(1-\alpha),$$

where $w:\mathbb{R}\to\mathbb{R}$ is defined as

$$w(\theta)=\sin((1+\alpha)\theta)\cos(\alpha\omega)/(1+\alpha)-\cos((1+\alpha)\theta)$$
$$-\sin((1-\alpha)\theta)\cos(\alpha\omega)/(1-\alpha)+\cos((1-\alpha)\theta),$$

$\omega=3\pi/2$ and $\alpha=0.54448373$ the approximated positive solution of the equation $\alpha\sin(2\omega)+\sin(2\alpha\omega)=0$. Using the denotations as in 2.2.1 the problem input data for this problem read as follows. For their computation one needs an alternative `cart2pol` function, namely `cart2polALT`, which maps the angle into the interval $[0,2\pi)$ instead of $(-\pi,\pi]$.

```
function val = w(theta)
omega = 3*pi/2;
alpha = .54448373;
val = (sin((1+alpha)*theta)*cos(alpha*omega)) / (1+alpha) - ...
       cos((1+alpha)*theta) - (sin((1-alpha)*theta)*cos(alpha*omega))/...
                                    (1-alpha) + cos((1-alpha)*theta);
end

function val = w1(theta)
omega = 3*pi/2;
alpha = .54448373;
val = ((1+alpha)*cos((1+alpha)*theta)*cos(alpha*omega)) / (1+alpha) + ...
                          (1+alpha)*sin((1+alpha)*theta) - (1-alpha)*...
                  (cos((1-alpha)*theta)*cos(alpha*omega))/(1-alpha) - ...
                                       (1-alpha)*sin((1-alpha)*theta);
end

function val = w2(theta)
omega = 3*pi/2;
alpha = .54448373;
val = (-(1+alpha)^2*sin((1+alpha)*theta)*cos(alpha*omega)) / (1+alpha) +...
               (1+alpha)^2*cos((1+alpha)*theta)+ ((1-alpha)^2*...
                   (sin((1-alpha)*theta)*cos(alpha*omega))/...
                      (1-alpha)) - (1-alpha)^2*cos((1-alpha)*theta);
end

function val = w3(theta)
omega = 3*pi/2;
```

```
alpha = .54448373;
val = (-(1+alpha)^3*cos((1+alpha)*theta)*cos(alpha*omega)) / (1+alpha) -...
                  (1+alpha)^3*sin((1+alpha)*theta) + ((1-alpha)^3*...
                  (cos((1-alpha)*theta)*cos(alpha*omega))/(1-alpha)) +...
                                    (1-alpha)^3*sin((1-alpha)*theta);
end

function val = uExact1(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = r.^alpha .* ((1+alpha)*sin(theta).*w(theta) + cos(theta) .* ...
                                                        w1(theta));
end

function val = uExact2(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
   val = r.^alpha .* (-(1+alpha)*cos(theta).*w(theta) + sin(theta) .* ...
                                                           w1(theta));
end

function val = f(x)
val = zeros(size(x));
end

function val = u4Db(x)
val = [uExact1(x) uExact2(x)];
end

function val = g(x)
val = zeros(size(x));
end

function val = gradExact1(x)
% omega = 3*pi/2;
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = - sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du1Dr = alpha * uExact1(x) ./r;
Du1Dtheta = r.^alpha .* ((1+alpha)*(cos(theta).*w(theta) + sin(theta).* ...
               w1(theta))-sin(theta).*w1(theta)+cos(theta).*w2(theta));
val(:,1) = DrDx .* Du1Dr + DthetaDx .*Du1Dtheta;
val(:,2) = DrDy .* Du1Dr + DthetaDy .*Du1Dtheta;
end

function val = gradExact2(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
```

```
DthetaDx = -sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du2Dr = alpha * uExact2(x) ./r;
Du2Dtheta = r.^alpha .* ((1+alpha)*sin(theta).*w(theta) - ...
           (1+alpha)*cos(theta).*w1(theta) + cos(theta) .* w1(theta) + ...
                                             sin(theta) .* w2(theta));
val(:,1) = DrDx .* Du2Dr + DthetaDx .*Du2Dtheta;
val(:,2) = DrDy .* Du2Dr + DthetaDy .*Du2Dtheta;
end

function val = Du4Db1(x)
val = gradExact1(x);
end

function val = Du4Db2(x)
val = gradExact2(x);
end

function val = pExact(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
val=-r.^(alpha-1).*((1+alpha)^2.*w1(theta) + w3(theta))/(1-alpha);
end

function val = sigmaExact1(x)
  val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
end

function val = sigmaExact2(x)
  val = gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
end

function [theta r] = cart2polALT(x,y)
% modified cart2pol for the angle convention
% theta in [0,2pi] instaed of [-pi,pi]
[theta,r] = cart2pol(x,y);
[ind] = theta<0;
theta(ind) = theta(ind)+2*pi;
end
```

The realisation of the AFEM cycle and the plots at the end of the program are identical to those from the previous subsection 2.2.1. The following figures show an example output of `afemCRP0StokesLshapeExact`.

Figure 2.2.4: Convergence behaviour of errors plotted by `afemCRP0StokesLshapeExact` with `minNrDof = 5 000` on the L-shape domain and initial mesh `Lshape`

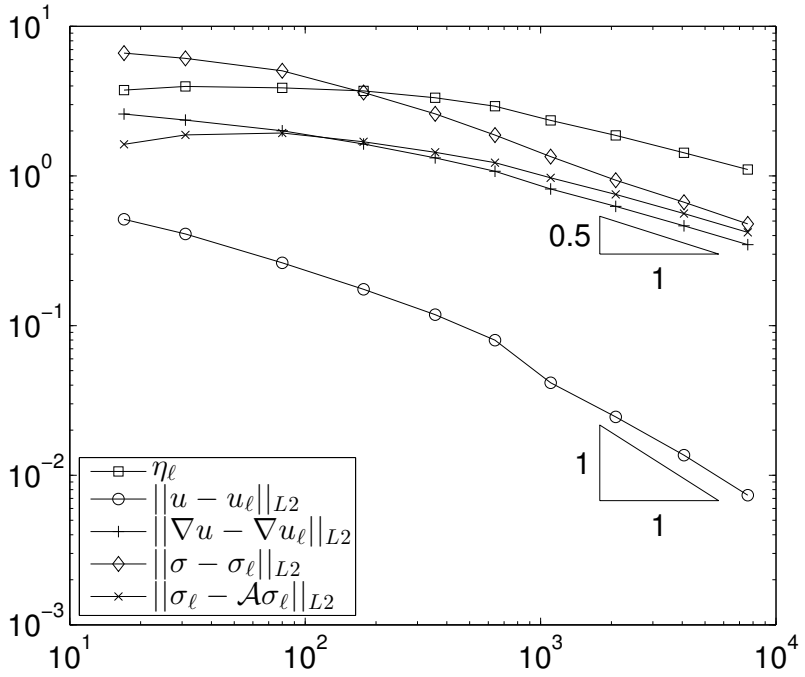


Figure 2.2.5: Mesh with 1 007 nodes plotted by `afemCRP0StokesLshapeExact` with `minNrDof = 5 000` on the L-shape domain and initial mesh `Lshape`

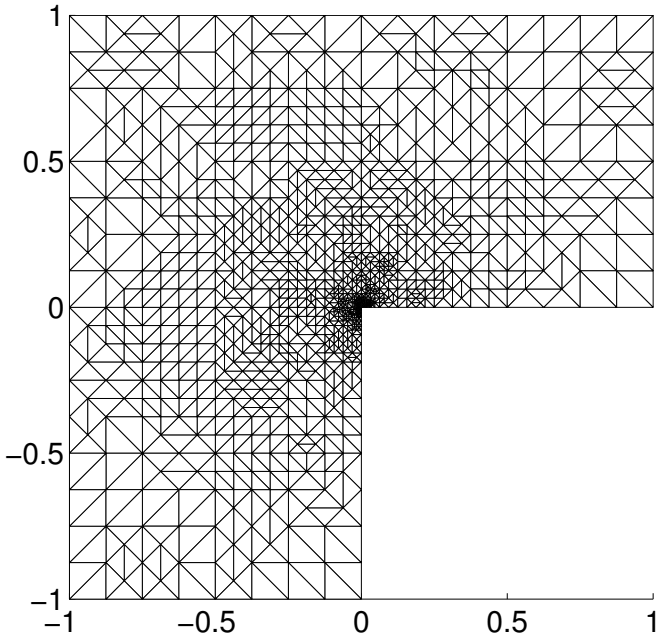
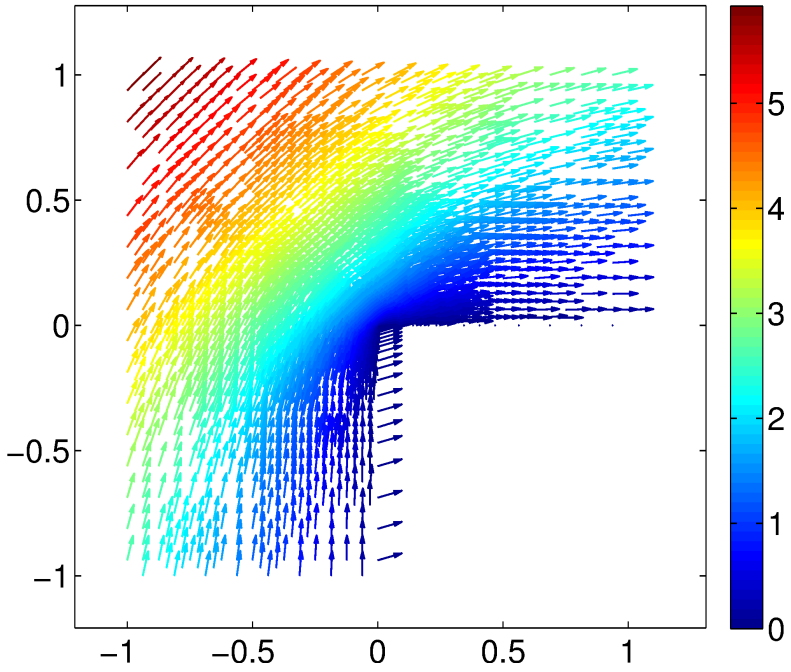


Figure 2.2.6: Solution for `u` with 7 594 degrees of freedom plotted by `afemCRP0StokesLshapeExact` with `minNrDof` = 5 000 on the L-shape domain and initial mesh `Lshape`



2.2.3 The Slit Example

The example program `afemCRP0StokesSlitExact.m` implements the AFEM cycle for the Stokes' equations on the slit domain $(-1,1) \times (-1,1) \setminus [0,1] \times \{0\}$ with the right-hand side $f \equiv 0$. If $(r,\theta) \in [0,\infty) \times [0,2\pi)$ are the polar coordinates of $x \in \overline{\Omega}$, the exact solution of this problem reads

$$u(x) = u(r,\theta) = 3r^{1/2}/2 \begin{pmatrix} \cos(\theta/2) - \cos(3\theta/2) \\ 3\sin(\theta/2) - \sin(3\theta/2) \end{pmatrix} \quad \text{and}$$
$$p(x) = p(r,\theta) = 6r^{-1/2}\cos(\theta/2).$$

As in 2.2.2 one needs an additional function for the transformation into polar coordinates. Then the problem input data are computed as below.

```
function val = uExact1(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = 1.5*sqrt(r).*(cos(0.5*theta)-cos(1.5*theta));
end

function val = uExact2(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = 1.5*sqrt(r).*(3*sin(0.5*theta)-sin(1.5*theta));
end

function val = f(x)
val = zeros(size(x));
end

function val = u4Db(x)
val = [uExact1(x) uExact2(x)];
end

function val = g(x)
val = zeros(size(x));
end

function val = gradExact1(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = - sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du1Dr = 0.75*(r.^-0.5).*(cos(0.5*theta)-cos(1.5*theta));
Du1Dtheta = 0.75*sqrt(r).*(-sin(0.5*theta)+3*sin(1.5*theta));
val(:,1) = DrDx .* Du1Dr + DthetaDx .*Du1Dtheta;
val(:,2) = DrDy .* Du1Dr + DthetaDy .*Du1Dtheta;
end

function val = gradExact2(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = -sin(theta)./r;
```

```
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du2Dr = 0.75*(r.^-0.5).*(3*sin(0.5*theta)-sin(1.5*theta));
Du2Dtheta = 2.25*sqrt(r).*(cos(0.5*theta)-cos(1.5*theta));
val(:,1) = DrDx .* Du2Dr + DthetaDx .*Du2Dtheta;
val(:,2) = DrDy .* Du2Dr + DthetaDy .*Du2Dtheta;
end

function val = Du4Db1(x)
val = gradExact1(x);
end

function val = Du4Db2(x)
val = gradExact2(x);
end

function val = pExact(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = -6*(r.^-0.5).*cos(0.5*theta);
end

function val = sigmaExact1(x)
val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
end

function val = sigmaExact2(x)
val=gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
end

function [theta r] = cart2polALT(x,y)
% modified cart2pol for the angle convention
% theta in [0,2pi] instaed of [-pi,pi]
[theta,r] = cart2pol(x,y);
[ind] = theta<0;
theta(ind) = theta(ind)+2*pi;
end
```

For the realisation of the AFEM cycle and the plots at the end of the program the reader is referred to 2.2.1. The following figures show an example output of the program `afemCRP0StokesSlitExact`.

Figure 2.2.7: Convergence behaviour of errors plotted by `afemCRP0StokesSlitExact` with `minNrDof = 5 000` on the slit domain and initial mesh `Slit`

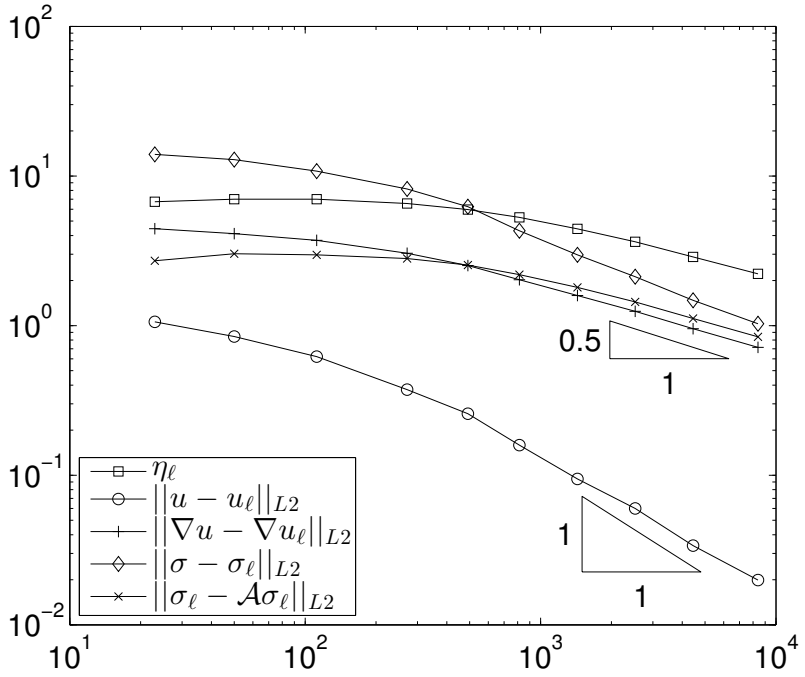


Figure 2.2.8: Mesh with 1 103 nodes plotted by `afemCRP0StokesSlitExact` with `minNrDof = 5 000` on the slit domain and initial mesh `Slit`

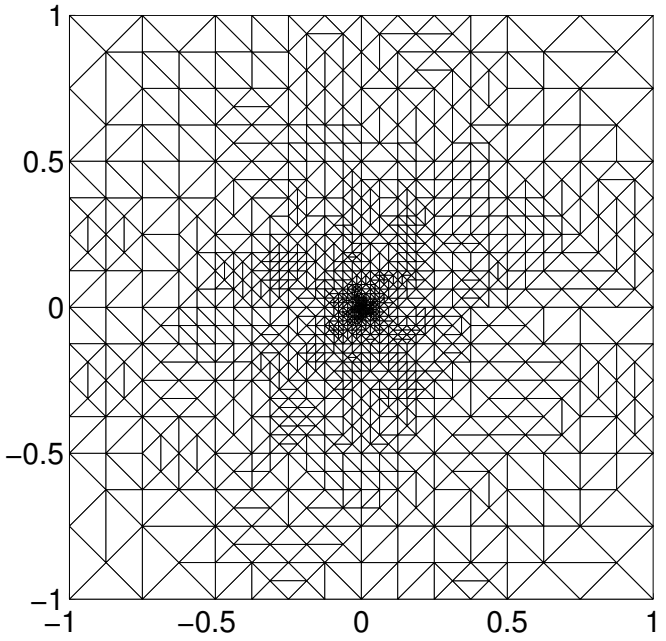
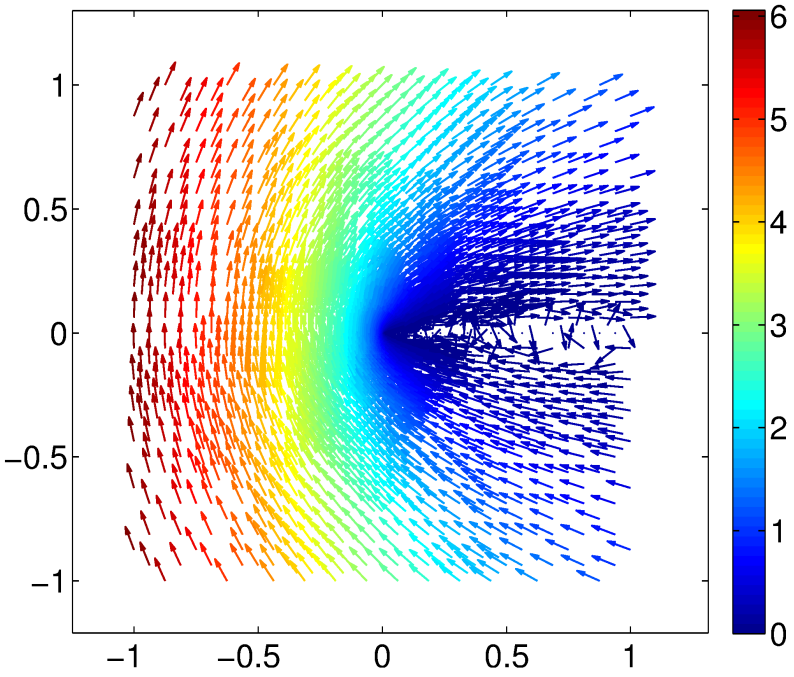


Figure 2.2.9: Solution for `u` with 8 387 degrees of freedom plotted by `afemCRP0StokesSlitExact` with `minNrDof = 5 000` on the slit domain and initial mesh `Slit`



2.2.4 The Backward Facing Step Example

For the backward facing step problem on the domain $\Omega = (-2, 8) \times (-1, 1) \setminus (-2, 0) \times (-1, 0)$ with the right-hand side $f \equiv 0$ and given Dirichlet boundary datum for $x \in \partial\Omega$

$$g(x) = \begin{cases} (0,0)^\top & \text{if } -2 < x_1 < 8 \\ 1/10(-x_2(x_2-1),0)^\top & \text{if } x_1 = -2 \\ 1/80(-(x_2-1)(x_2+1),0)^\top & \text{if } x_1 = 8 \end{cases}$$

no exact solution is known. But an approximation is implemented in the example program `afemCRP0StokesBackwardFacingStep.m`. The problem input data with the denotations as in 2.2.1 read

```
function val = u4Db(x)
val = zeros(size(x));
for j=1:size(x,1)
    if (x(j,1) == -2)
        val(j,:) = [-x(j,2)*(x(j,2)-1)/10, 0];
    elseif (x(j,1) == 8)
        val(j,:) = [-(x(j,2)+1)*(x(j,2)-1)/80, 0];
    end
end
end

function val = Du4Db1(x)
val = zeros(size(x));
for j=1:size(x,1)
    if (x(j,1) == -2)
         val(j,:) = [0, (-2*x(j,2)+1)/10];
    elseif (x(j,1) == 8)
        val(j,:) = [0, -2*x(j,2)/80];
    end
end
end

function val = Du4Db2(x)
val = zeros(size(x));
end

function val = f(x)
val = zeros(size(x));
end

function val = g(x)
val = zeros(size(x));
end
```

Firstly the special initial triangulation is generated directly with some local refinements at the non-homogeneous Dirichlet boundary for $x_1 = -2$ and the arrays intended for the values of the error estimators and the exact errors are created.

```
    addpath(genpath(pwd));
```

```
    c4n = [0 -1;2 -1;4 -1;6 -1;8 -1; 8 1;6 1;4 1;2 1;0 1];
    n4e = [9 1 2;1 9 10;8 2 3;2 8 9;3 7 8;7 3 4;6 4 5;4 6 7];
    n4sDb = [1 2;2 3;3 4;4 5;5 6;6 7;7 8;8 9;9 10];
    n4sNb = zeros(0,2);

    [c4n,n4e,n4sDb,n4sNb] = refineUniformRed(c4n,n4e,n4sDb,n4sNb);
    c4n(28:31,:) = [-2 0;-1 0;-1 1;-2 1];
    n4e = [n4e;10 29 24;29 10 30;30 28 29;28 30 31];
    n4sDb = [n4sDb;29 24;24 1;28 29;31 28;30 31;10 30];

    eta4nrDoF = sparse(1,1);
    averaging4lvl = [];
    nrDoF4lvl = [];
    error4lvl = [];
    energy4lvl = [];
    stress4lvl = [];
```

The realisation of the AFEM loop and the commands for the plots of mesh, solution and convergence graphs are similar to those in 2.2.1 except that there are no exact errors in this program. Its example output can be found in the following figures.

Figure 2.2.10: Convergence behaviour of errors plotted by `afemCRP0BackwardFacingStep` with `minNrDof = 5 000` on the domain $\Omega = (-2, 8) \times (-1, 1) \setminus (-2, 0) \times (-1, 0)$ and initial mesh described in the above listing

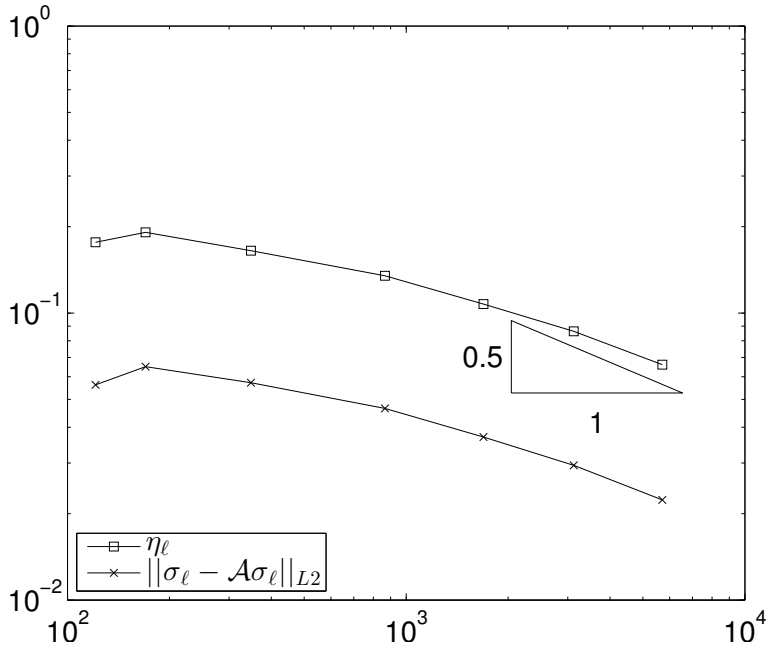


Figure 2.2.11: Mesh with 790 nodes plotted by `afemCRP0BackwardFacingStep` with `minNrDof = 5 000` on the domain $\Omega = (-2, 8) \times (-1, 1) \setminus (-2, 0) \times (-1, 0)$ and initial mesh described in the above listing

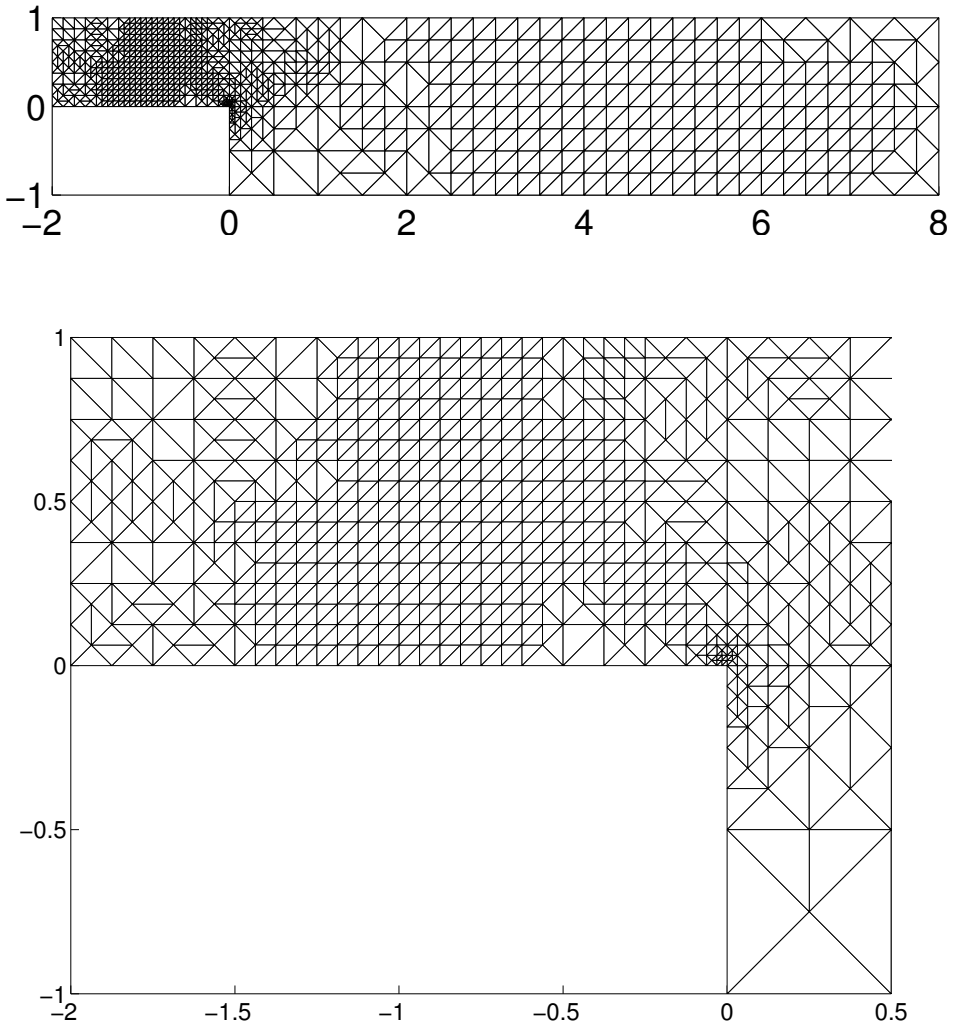
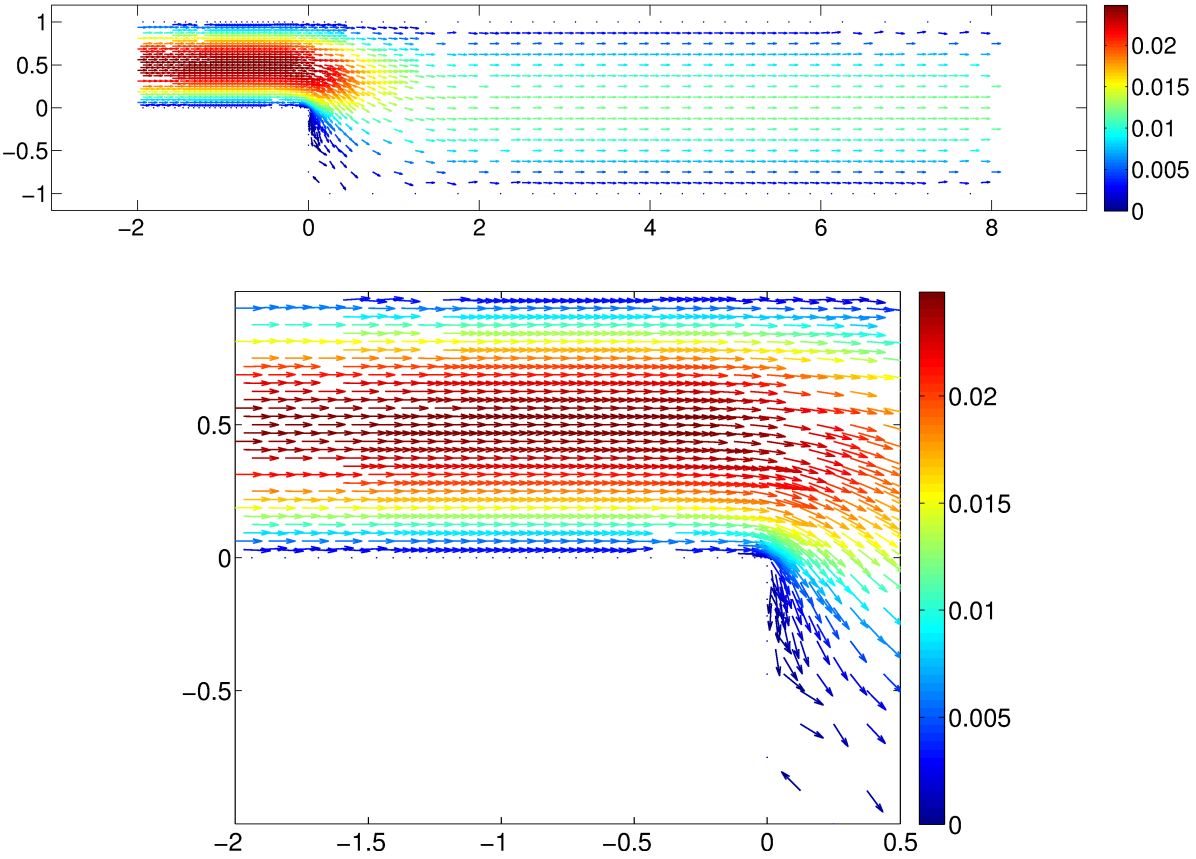


Figure 2.2.12: Solution for `u` with 5 703 degrees of freedom plotted by `afemCRP0BackwardFacingStep` with `minNrDof = 5 000` on the domain $\Omega = (-2, 8) \times (-1, 1) \setminus (-2, 0) \times (-1, 0)$ and initial mesh described in the above listing



3 Sources

3.1 List of Functions

3.2 Structure of the AFEM Directory

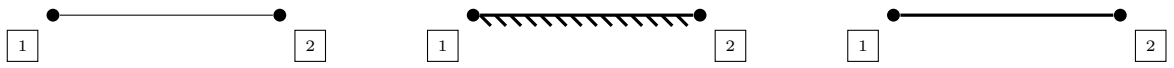
```
|-- afemCRP0StokesBackwardFacingStep.m
|-- afemCRP0StokesCollidingFlow.m
|-- afemCRP0StokesLshapeExact.m
|-- afemCRP0StokesSlitExact.m
|-- afemCRPoisson.m
|-- afemCRPoissonLShapeExact.m
```

```
|-- afemCRPoissonSquareExact.m       |   |   |-- BigSquare.m
|-- afemIntegrate.m                  |   |   |-- BigSquare_c4n.dat
|-- afemP1P1Elasticity.m             |   |   |-- BigSquare_n4e.dat
|-- afemP1P1ElasticityCookMembrane.m |   |   |-- BigSquare_n4sDb.dat
|-- afemP1P1ElasticitySquare.m       |   |   `-- BigSquare_n4sNb.dat
|-- afemP1P1ElasticitySquareExact.m  |   |-- CookMembrane
|-- afemP1P1ElasticitySquareNb.m     |   |   |-- CookMembrane_c4n.dat
|-- afemP1Poisson.m                  |   |   |-- CookMembrane_n4e.dat
|-- afemP1PoissonLShapeExact.m       |   |   |-- CookMembrane_n4sDb.dat
|-- afemP1PoissonShort.m             |   |   `-- CookMembrane_n4sNb.dat
|-- afemP1PoissonSlitExact.m         |   |-- Lshape
|-- afemP1PoissonSquareExact.m       |   |   |-- Lshape.m
|-- afemP1PoissonTeach.m             |   |   |-- Lshape_c4n.dat
|-- afemRT0Poisson.m                 |   |   |-- Lshape_n4e.dat
|-- afemRT0PoissonSlitExact.m        |   |   |-- Lshape_n4sDb.dat
|-- afemRT0PoissonSquareExact.m      |   |   `-- Lshape_n4sNb.dat
|-- common                           |   |-- LshabeNb
|   |-- computeArea4e.m              |   |   |-- LshapeNb.m
|   |-- computeArea4n.m              |   |   |-- LshapeNb_c4n.dat
|   |-- computeE4n.m                 |   |   |-- LshapeNb_n4e.dat
|   |-- computeE4s.m                 |   |   |-- LshapeNb_n4sDb.dat
|   |-- computeLength4s.m            |   |   `-- LshapeNb_n4sNb.dat
|   |-- computeMid4e.m               |   |-- LshapeRot
|   |-- computeMid4s.m               |   |   |-- LshapeRot.m
|   |-- computeN4s.m                 |   |   |-- LshapeRot_c4n.dat
|   |-- computeNormal4e.m            |   |   |-- LshapeRot_n4e.dat
|   |-- computeNormal4s.m            |   |   |-- LshapeRot_n4sDb.dat
|   |-- computeS4e.m                 |   |   `-- LshapeRot_n4sNb.dat
|   |-- computeS4n.m                 |   |-- Slit
|   |-- computeTangent4e.m           |   |   |-- Slit.m
|   |-- computeTangent4s.m           |   |   |-- Slit_c4n.dat
|   |-- loadGeometry.m               |   |   |-- Slit_n4e.dat
|   |-- P0AveragingP1.m              |   |   |-- Slit_n4sDb.dat
|   |-- P0NormalJump.m               |   |   `-- Slit_n4sNb.dat
|   `-- P0TangentJump.m              |   |-- SlitNb
|-- estimate                         |   |   |-- SlitNb.m
|   |-- estimateCREtaNormalOnly.m    |   |   |-- SlitNb_c4n.dat
|   |-- estimateCREtaSides.m         |   |   |-- SlitNb_n4e.dat
|   |-- estimateCRP0Elements.m       |   |   |-- SlitNb_n4sDb.dat
|   |-- estimateL2Averaging.m        |   |   `-- SlitNb_n4sNb.dat
|   |-- estimateNCEtaElements.m      |   |-- Square
|   |-- estimateP1AveragingP1.m      |   |   |-- Square.m
|   |-- estimateP1EtaElements.m      |   |   |-- Square_c4n.dat
|   |-- estimateP1EtaSides.m         |   |   |-- Square_n4e.dat
|   |-- estimateRT0EtaSides.m        |   |   |-- Square_n4sDb.dat
|   `-- estimateSigmaAveragingP1.m   |   |   `-- Square_n4sNb.dat
|-- extern                           |   |-- SquareNb
|   `-- quiver2                      |   |   |-- SquareNb.m
|       |-- license.txt              |   |   |-- SquareNb_c4n.dat
|       `-- quiver2.m                |   |   |-- SquareNb_n4e.dat
|-- geometries                       |   |   |-- SquareNb_n4sDb.dat
|   |-- BigSquare                    |   |   `-- SquareNb_n4sNb.dat
```

```
|   |-- SquareNb2                    |   |-- plotConvergence.m
|   |   |-- SquareNb2.m              |   |-- plotCR.m
|   |   |-- SquareNb2_c4n.dat        |   |-- plotCRP0.m
|   |   |-- SquareNb2_n4e.dat        |   |-- plotEtaSidesTeach.m
|   |   |-- SquareNb2_n4sDb.dat      |   |-- plotP04e.m
|   |   `-- SquareNb2_n4sNb.dat      |   |-- plotP04s.m
|   |-- Stokes                       |   |-- plotP1.m
|   |   |-- Stokes_c4n.dat           |   |-- plotP1P1.m
|   |   |-- Stokes_n4e.dat           |   |-- plotRT0.m
|   |   |-- Stokes_n4sDb.dat         |   `-- plotTriangulation.m
|   |   `-- Stokes_n4sNb.dat         |-- pool
|   `-- Tree                         |   |-- AW
|       |-- Tree_c4n.dat             |   |   |-- afemAWElasticityTest.m
|       |-- Tree_n4e.dat             |   |   `-- solveAWElasticityTest.m
|       |-- Tree_n4sDb.dat           |   |-- P1
|       `-- Tree_n4sNb.dat           |   |   |-- computeP1Grad.m
|-- integrate                        |   |   `-- evaluateP1.m
|   |-- error4eCREnergy.m            |   `-- drawTriangle.m
|   |-- error4eCRL2.m                |-- refine
|   |-- error4eP1Energy.m            |   |-- closure.m
|   |-- error4eP1L2.m                |   |-- refineBi3GB.m
|   |-- error4eRT0Energy.m           |   |-- refineBi5GB.m
|   |-- error4eRT0L2.m               |   |-- refineRGB.m
|   |-- error4eStokesCRStress.m      |   `-- refineUniformRed.m
|   |-- integrate.m                  `-- solve
|   |-- L2Norm.m                         |-- solveAWElasticity.m
|   `-- oscillations.m                   |-- solveCRP0Stokes.m
|-- mark                                 |-- solveCRPoisson.m
|   |-- markBulk.m                       |-- solveCRPoissonNEU.m
|   |-- markMaximum.m                    |-- solveP1P1Elasticity.m
|   `-- markUniform.m                    |-- solveP1Poisson.m
|-- plot                                 `-- solveRT0Poisson.m
```

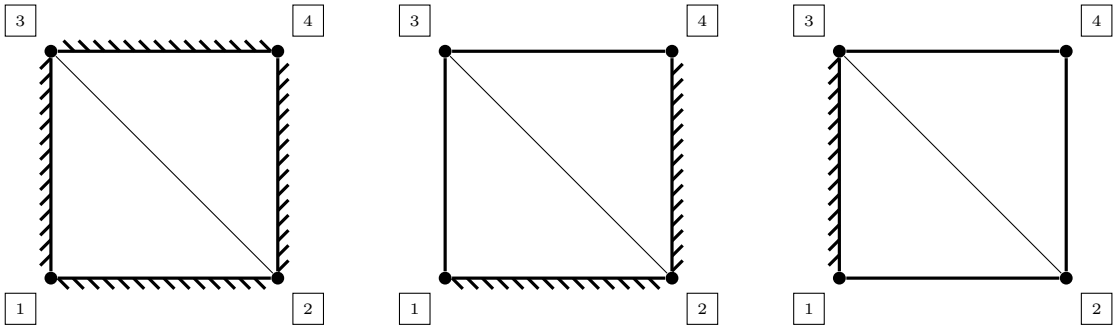
3.3 Geometries

Figure 3.3.1: Legend



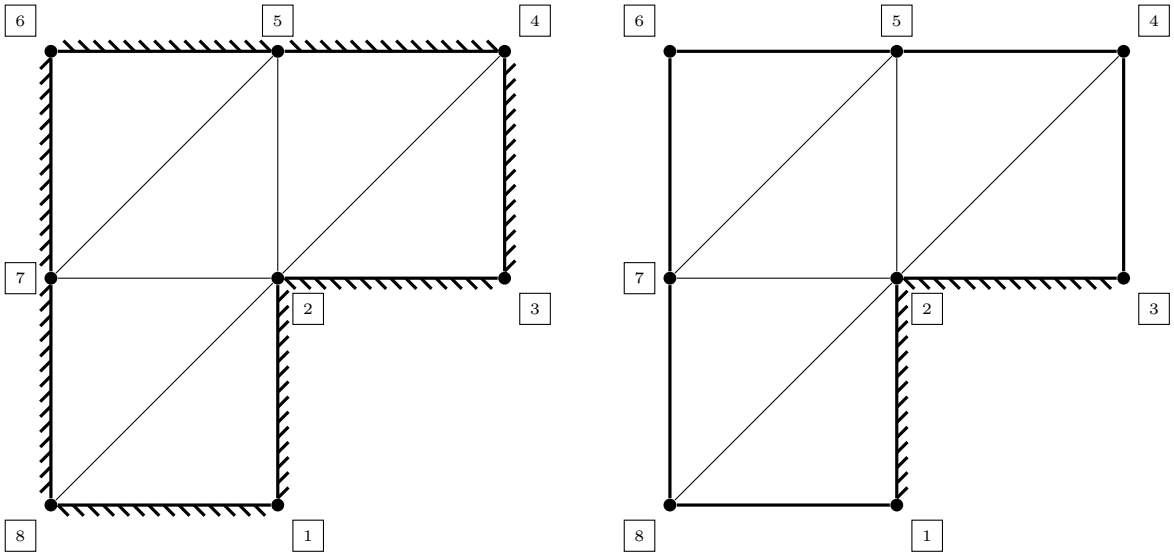
(a) two nodes and one side (b) Dirichlet boundary side (c) Neumann boundary side

Figure 3.3.2: Geometries on $\Omega = (0,1)^2$



(a) Square geometry (b) SquareNb geometry (c) SquareNb2 geometry

Figure 3.3.3: Geometries on $\Omega = (-1,1)^2 \setminus [0,1] \times [-1,0]$



(a) LShape geometry (b) LShapeNb geometry

Figure 3.3.4: LShapeRot geometry on the rotated L-Shape 3.3.3

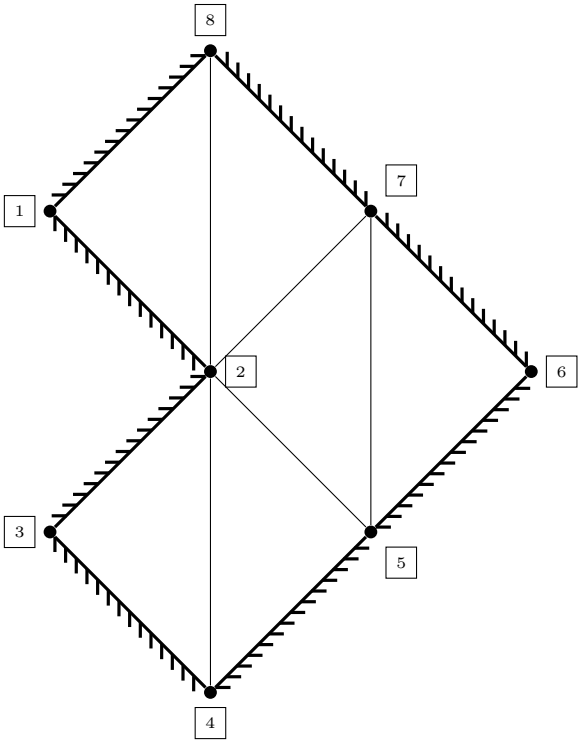


Figure 3.3.5: SlitNb geometry on $\Omega = (-1,1)^2 \setminus [0,1] \times \{0\}$

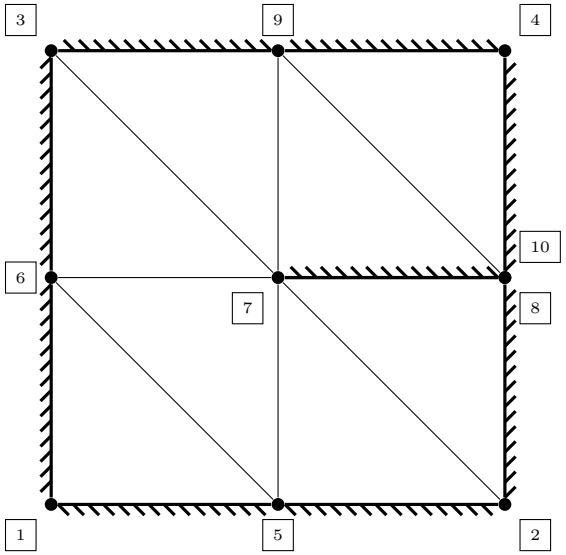
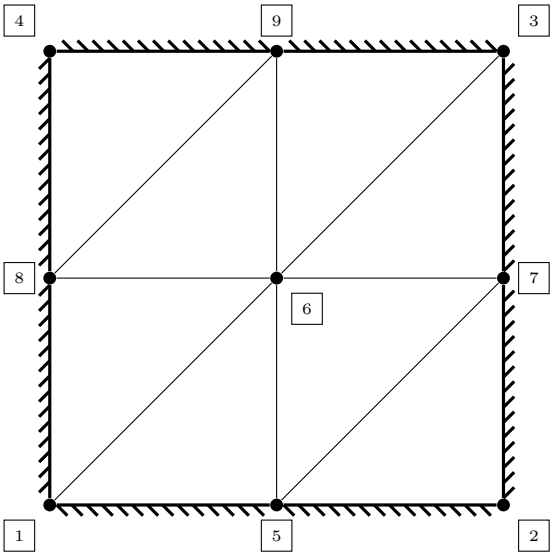


Figure 3.3.6: BigSquare geometry on the big unit square $\Omega = (-1,1)^2$



3.4 Sources

Listing 3.1: afemCRP0StokesBackwardFacingStep.m

```
function afemCRP0StokesBackwardFacingStep
%% afemCRP0StokesBackwardFacingStep - Solve Stokes problem with CR elements
% Solve the backward facing step example of the Stokes problem with linear
% CR finite elements adaptively. The colliding flow example is considered
% on the domain (-2,8)x(-1,2)\(-2,0)x(-1,0) with constant right-hand side
% f=0 and full Dirichlet boundary condition.
% An exact solution is NOT known.

    %% Parameters
    minNrDoF = 20000; % minimal number of degrees of freedom
    theta = 0.5; % bulk parameter (theta = 1 for uniform refinement)

    %% Initialization
    addpath(genpath(pwd));
    c4n = [0 -1;2 -1;4 -1;6 -1;8 -1; 8 1;6 1;4 1;2 1;0 1];
    n4e = [9 1 2;1 9 10;8 2 3;2 8 9;3 7 8;7 3 4;6 4 5;4 6 7];
    n4sDb = [1 2;2 3;3 4;4 5;5 6;6 7;7 8;8 9;9 10];
    n4sNb = zeros(0,2);

    [c4n,n4e,n4sDb,n4sNb] = refineUniformRed(c4n,n4e,n4sDb,n4sNb);
    c4n(28:31,:) = [-2 0;-1 0;-1 1;-2 1];
    n4e = [n4e;10 29 24;29 10 30;30 28 29;28 30 31];
    n4sDb = [n4sDb;29 24;24 1;28 29;31 28;30 31;10 30];

    eta4nrDoF = sparse(1,1);
    averaging4lvl = [];
    nrDoF4lvl = [];
    error4lvl = [];
    energy4lvl = [];
    stress4lvl = [];

    %% AFEM loop
    while( true )
        %% SOLVE
        [u,p,~,~,nrDoF,gradU4e] = solveCRP0Stokes(@(x)f(x),@(x)u4Db(x),...
                                        @(x)g(x),c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        % Averaging estimated error
        averaging4lvl(end+1) = ...
               sqrt(sum(estimateSigmaAveragingP1(c4n,n4e,[...
                   reshape(gradU4e(1,:,:),2,size(gradU4e,3))' - ...
                                                [p,zeros(size(p))],...
                   reshape(gradU4e(2,:,:),2,size(gradU4e,3))' - ...
                                                [zeros(size(p)),p]])));
        %% ESTIMATE
        [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,...
                       @(x)f(x),@(x)Du4Db1(x),@(x)Du4Db2(x),u,gradU4e);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4e));
```

```
        % print information on afem loop
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        % break condition
        if nrDoF >= minNrDoF, break, end;
        %% MARK
        if theta==1
            n4sMarked = markUniform(n4e);
        else
            n4sMarked = markBulk(n4e,eta4e,theta);
        end
        %% REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    %% Plot mesh, solution and convergence graph
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    % Mesh
    figure;
    plotTriangulation(c4n,n4e);
    % Solution
    figure;
    plotCRP0(c4n,n4e,u,p);
    % Convergence graphs
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,averaging4lvl,'||\sigma_l - A \sigma_l||_{L2}');
end


%% PROBLEM INPUT DATA
function val = u4Db(x)
val = zeros(size(x,1),2);
ind1 = x(:,1) == -2;
ind2 = x(:,1) == 8;
val(ind1,1) = -x(ind1,2).*(x(ind1,2)-1)/10;
val(ind2,1) = -(x(ind2,2)+1).*(x(ind2,2)-1)/80;
end

function val = Du4Db1(x)
val = zeros(size(x,1),2);
ind1 = x(:,1) == -2;
ind2 = x(:,1) == 8;
% du1/dx2 on left and right boundary
val(ind1,2) = (-2*x(ind1,2)+1)/10;
val(ind2,2) = -2*x(ind2,2)/80;
end

function val = Du4Db2(x)
val = zeros(size(x));
```

```
end

function val = f(x)
val = zeros(size(x));
end

function val = g(x)
val = zeros(size(x));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.2: afemCRP0StokesCollidingFlow.m

```
function afemCRP0StokesCollidingFlow
%% afemCRP0StokesCollidingFlow - Solve Stokes problem with CR elements.
% Solve the colliding flow example of the Stokes problem with linear
% CR finite elements adaptively. The colliding flow example is considered
% on the unit square (-1,1)^2 with constant right-hand side f=0 and full
% Dirichlet boundary condition. An exact solution is known.

    %% Parameters
    minNrDoF = 5000; % minimal number of degrees of freedom
    theta = 0.5; % bulk parameter (theta = 1 for uniform refinement)

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('BigSquare',1);
    eta4nrDoF = sparse(1,1);
    averaging4lvl = [];
```

```
nrDoF4lvl = [];
error4lvl = [];
energy4lvl = [];
stress4lvl = [];

%% AFEM loop
while( true )
    %% SOLVE
    [u,p,~,~,nrDoF,gradU4e] = solveCRP0Stokes(@(x)f(x),@(x)u4Db(x),...
                                    @(x)g(x),c4n,n4e,n4sDb,n4sNb);
    nrDoF4lvl(end+1) = nrDoF;
    gradU4e1 = reshape(gradU4e(1,:,:),2,size(gradU4e,3))';
    gradU4e2 = reshape(gradU4e(2,:,:),2,size(gradU4e,3))';
    % Averaging estimated error
    averaging4lvl(end+1) = ...
           sqrt(sum(estimateSigmaAveragingP1(c4n,n4e,[gradU4e1 - ...
               [p,zeros(size(p))],gradU4e2 - [zeros(size(p)),p]])));
    % L2 exact error
    error4lvl(end+1) = ...
           sqrt(sum(error4eCRL2(c4n,n4e,@(x)uExact1(x),u(:,1)))+...
                  sum(error4eCRL2(c4n,n4e,@(x)uExact2(x),u(:,2))));
    % Energy exact error
    energy4lvl(end+1) = ...
        sqrt(sum(error4eCREnergy(c4n,n4e,@(x)gradExact1(x),u(:,1)))+...
           sum(error4eCREnergy(c4n,n4e,@(x)gradExact2(x),u(:,2))));
    % Stress exact error
    stress4lvl(end+1) = ...
                  sqrt(sum(error4eStokesCRStress(c4n,n4e,1,...
                         @(x)sigmaExact1(x),u(:,1),p,gradU4e1))+...
                      sum(error4eStokesCRStress(c4n,n4e,2,...
                         @(x)sigmaExact2(x),u(:,2),p,gradU4e2)));
    %% ESTIMATE
    [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,...
                  @(x)f(x),@(x)Du4Db1(x),@(x)Du4Db2(x),u,gradU4e);
    eta4nrDoF(nrDoF) = sqrt(sum(eta4e));
    % print information on afem loop
    disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
        '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
    % break condition
    if nrDoF >= minNrDoF, break, end;
    %% MARK
    if theta==1
        n4sMarked = markUniform(n4e);
    else
        n4sMarked = markBulk(n4e,eta4e,theta);
    end
    %% REFINE
    [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
end

%% Plot mesh, solution and convergence graph
nrDoF4lvl = find(eta4nrDoF);
```

```
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    % Mesh
    figure;
    plotTriangulation(c4n,n4e);
    % Solution
    figure;
    plotCRP0(c4n,n4e,u,p);
    % Convergence graphs
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u-u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nabla u - \nabla u_l||_{L2}');
    plotConvergence(nrDoF4lvl,stress4lvl,'||\sigma - \sigma_l||_{L2}');
    plotConvergence(nrDoF4lvl,averaging4lvl,'||\sigma_l - A \sigma_l||_{L2}');
end


%% PROBLEM INPUT DATA
function val = uExact1(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = 20*x1.*x2.^4-4*x1.^5;
end

function val = uExact2(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = 20*x1.^4.*x2 - 4*x2.^5;
end

function val = f(x)
    val = zeros(size(x));
end

function val = u4Db(x)
    val = [uExact1(x),uExact2(x)];
end

function val = g(x)
    val = zeros(size(x));
end

function val = gradExact1(x)
    val(:,1) = 20*x(:,2).^4-20*x(:,1).^4;
    val(:,2) = 80*x(:,1).*x(:,2).^3;
end

function val = gradExact2(x)
    val (:,1) = 80*x(:,1).^3.*x(:,2);
    val(:,2) = 20*x(:,1).^4-20*x(:,2).^4;
end
```

```
121
    function val = Du4Db1(x)
       val = gradExact1(x);
    end

126 function val = Du4Db2(x)
       val = gradExact2(x);
    end

    function val = pExact(x)
131    x1 = x(:,1);
       x2 = x(:,2);
       val = (120*x1.^2.*x2.^2-20*x1.^4-20*x2.^4-32/6);
    end

136 function val = sigmaExact1(x)
       val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
    end

    function val = sigmaExact2(x)
141    val = gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Copyright 2009-2015
146 % Numerical Analysis Group
    % Prof. Dr. Carsten Carstensen
    % Humboldt-University
    % Departement of Mathematics
    % 10099 Berlin
151 % Germany
    %
    % This file is part of AFEM.
    %
    % AFEM is free software; you can redistribute it and/or modify
156 % it under the terms of the GNU General Public License as published by
    % the Free Software Foundation; either version 3 of the License, or
    % (at your option) any later version.
    %
    % AFEM is distributed in the hope that it will be useful,
161 % but WITHOUT ANY WARRANTY; without even the implied warranty of
    % MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    % GNU General Public License for more details.
    %
    % You should have received a copy of the GNU General Public License
166 % along with this program. If not, see <http://www.gnu.org/licenses/>.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.3: afemCRP0StokesLshapeExact.m

```
1  function afemCRP0StokesLshapeExact
   %% afemCRP0StokesLshapeExact - Solve Stokes problem with CR elements.
3  % Solve the Stokes problem with linear CR finite elements adaptively on the
```

```
% L-shape domain with constant right-hand side f=0 and full Dirichlet
% boundary condition. An exact solution is known.

    %% Parameters
    minNrDoF = 5000; % minimal number of degrees of freedom
    theta = 0.5; % bulk parameter (theta = 1 for uniform refinement)

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('Lshape',0);
    eta4nrDoF = sparse(1,1);
    averaging4lvl = [];
    nrDoF4lvl = [];
    error4lvl = [];
    energy4lvl = [];
    stress4lvl = [];

    %% AFEM loop
    while( true )
        %% SOLVE
        [u,p,~,~,nrDoF,gradU4e] = solveCRP0Stokes(@(x)f(x),@(x)u4Db(x),...
                                        @(x)g(x),c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        gradU4e1 = reshape(gradU4e(1,:,:),2,size(gradU4e,3))';
        gradU4e2 = reshape(gradU4e(2,:,:),2,size(gradU4e,3))';
        % Averaging estimated error
        averaging4lvl(end+1) = ...
               sqrt(sum(estimateSigmaAveragingP1(c4n,n4e,[gradU4e1 - ...
                   [p,zeros(size(p))],gradU4e2 - [zeros(size(p)),p]])));
        % L2 exact error
        error4lvl(end+1) = ...
               sqrt(sum(error4eCRL2(c4n,n4e,@(x)uExact1(x),u(:,1)))+...
                      sum(error4eCRL2(c4n,n4e,@(x)uExact2(x),u(:,2))));
        % Energy exact error
        energy4lvl(end+1) = ...
            sqrt(sum(error4eCREnergy(c4n,n4e,@(x)gradExact1(x),u(:,1)))+...
               sum(error4eCREnergy(c4n,n4e,@(x)gradExact2(x),u(:,2))));
        % Stress exact error
        stress4lvl(end+1) = ...
                      sqrt(sum(error4eStokesCRStress(c4n,n4e,1,...
                             @(x)sigmaExact1(x),u(:,1),p,gradU4e1))+...
                         sum(error4eStokesCRStress(c4n,n4e,2,...
                             @(x)sigmaExact2(x),u(:,2),p,gradU4e2)));
        %% ESTIMATE
        [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,...
                      @(x)f(x),@(x)Du4Db1(x),@(x)Du4Db2(x),u,gradU4e);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4e));
        % print information on afem loop
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        % break condition
        if nrDoF >= minNrDoF, break, end;
```

```
        %% MARK
        if theta==1
            n4sMarked = markUniform(n4e);
        else
            n4sMarked = markBulk(n4e,eta4e,theta);
        end
        %% REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    %% Plot mesh, solution and convergence graph
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    % Mesh
    figure;
    plotTriangulation(c4n,n4e);
    % Solution
    figure
    plotCRP0(c4n,n4e,u,p);
    % Convergence graphs
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u-u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nabla u - \nabla u_l||_{L2}');
    plotConvergence(nrDoF4lvl,stress4lvl,'||\sigma - \sigma_l||_{L2}');
    plotConvergence(nrDoF4lvl,averaging4lvl,'||\sigma_l - A \sigma_l||_{L2}');
end


%% PROBLEM INPUT DATA
function val = w(theta)
omega = 3*pi/2;
alpha = .54448373;
val = (sin((1+alpha)*theta)*cos(alpha*omega)) / (1+alpha) - ...
       cos((1+alpha)*theta) - (sin((1-alpha)*theta)*cos(alpha*omega))/...
                                    (1-alpha) + cos((1-alpha)*theta);
end

function val = w1(theta)
omega = 3*pi/2;
alpha = .54448373;
val = cos((1+alpha)*theta)*cos(alpha*omega) + ...
       (1+alpha)*sin((1+alpha)*theta) - ...
              cos((1-alpha)*theta)*cos(alpha*omega) - ...
                                     (1-alpha)*sin((1-alpha)*theta);
end

function val = w2(theta)
omega = 3*pi/2;
alpha = .54448373;
val = (1+alpha)*(-sin((1+alpha)*theta)*cos(alpha*omega) +...
```

```
108               (1+alpha)*cos((1+alpha)*theta))+ (1-alpha)*(...
                     (sin((1-alpha)*theta)*cos(alpha*omega))...
                               - (1-alpha)*cos((1-alpha)*theta));
    end

113 function val = w3(theta)
    omega = 3*pi/2;
    alpha = .54448373;
    val = -(1+alpha)^2*(cos((1+alpha)*theta)*cos(alpha*omega) +...
                      (1+alpha)*sin((1+alpha)*theta)) + (1-alpha)^2*(...
118                   cos((1-alpha)*theta)*cos(alpha*omega) +...
                                        (1-alpha)*sin((1-alpha)*theta));
    end

    function val = uExact1(x)
123 alpha = .54448373;
    [theta,r] = cart2polALT(x(:,1),x(:,2));
    val = r.^alpha .* ((1+alpha)*sin(theta).*w(theta) + cos(theta) .* ...
                                                          w1(theta));
    end
128
    function val = uExact2(x)
    alpha = .54448373;
    [theta,r] = cart2polALT(x(:,1),x(:,2));
       val = r.^alpha .* (-(1+alpha)*cos(theta).*w(theta) + sin(theta) .* ...
133                                                          w1(theta));
    end

    function val = f(x)
    val = zeros(size(x));
138 end

    function val = u4Db(x)
    val = [uExact1(x) uExact2(x)];
    end
143
    function val = g(x)
    val = zeros(size(x));
    end

148 function val = gradExact1(x)
    % omega = 3*pi/2;
    alpha = .54448373;
    [theta,r] = cart2polALT(x(:,1),x(:,2));
    DrDx = cos(theta);
153 DthetaDx = - sin(theta)./r;
    DrDy=sin(theta);
    DthetaDy = cos(theta)./r;
    Du1Dr = alpha * uExact1(x) ./r;
    Du1Dtheta = r.^alpha .* ((1+alpha)*(cos(theta).*w(theta) + sin(theta).* ...
158                 w1(theta))-sin(theta).*w1(theta)+cos(theta).*w2(theta));
    val(:,1) = DrDx .* Du1Dr + DthetaDx .*Du1Dtheta;
```

```
val(:,2) = DrDy .* Du1Dr + DthetaDy .*Du1Dtheta;
end

function val = gradExact2(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = -sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du2Dr = alpha * uExact2(x) ./r;
Du2Dtheta = r.^alpha .* ((1+alpha)*sin(theta).*w(theta) - ...
          (1+alpha)*cos(theta).*w1(theta) + cos(theta) .* w1(theta) + ...
                                            sin(theta) .* w2(theta));
val(:,1) = DrDx .* Du2Dr + DthetaDx .*Du2Dtheta;
val(:,2) = DrDy .* Du2Dr + DthetaDy .*Du2Dtheta;
end

function val = Du4Db1(x)
val = gradExact1(x);
end

function val = Du4Db2(x)
val = gradExact2(x);
end

function val = pExact(x)
alpha = .54448373;
[theta,r] = cart2polALT(x(:,1),x(:,2));
val=-r.^(alpha-1).*((1+alpha)^2.*w1(theta) + w3(theta))/(1-alpha);
end

function val = sigmaExact1(x)
  val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
end

function val = sigmaExact2(x)
  val = gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
end

function [theta r] = cart2polALT(x,y)
% modified cart2pol for the angle convention
% theta in [0,2pi] instead of [-pi,pi]
[theta,r] = cart2pol(x,y);
[ind] = theta<0;
theta(ind) = theta(ind)+2*pi;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
```

```
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.4: afemCRP0StokesSlitExact.m

```
function afemCRP0StokesSlitExact
%% afemCRP0StokesSlitExact - Solve Stokes problem with CR elements.
% Solve the Stokes problem with linear CR finite elements adaptively on the
% slit domain with constant right-hand side f=0 and full Dirichlet
% boundary condition. An exact solution is known.

    %% Parameters
    minNrDoF = 5000; % minimal number of degrees of freedom
    theta = 0.5; % bulk parameter (theta = 1 for uniform refinement)

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('Slit',0);
    eta4nrDoF = sparse(1,1);
    averaging4lvl = [];
    nrDoF4lvl = [];
    error4lvl = [];
    energy4lvl = [];
    stress4lvl = [];

    %% AFEM loop
    while( true )
        %% SOLVE
        [u,p,~,~,nrDoF,gradU4e] = solveCRP0Stokes(@(x)f(x),@(x)u4Db(x),...
                                        @(x)g(x),c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        gradU4e1 = reshape(gradU4e(1,:,:),2,size(gradU4e,3))';
        gradU4e2 = reshape(gradU4e(2,:,:),2,size(gradU4e,3))';
        % Averaging estimated error
        averaging4lvl(end+1) = ...
```

```
            sqrt(sum(estimateSigmaAveragingP1(c4n,n4e,[gradU4e1 - ...
                [p,zeros(size(p))],gradU4e2 - [zeros(size(p)),p]])));
        % L2 exact error
        error4lvl(end+1) = ...
            sqrt(sum(error4eCRL2(c4n,n4e,@(x)uExact1(x),u(:,1)))+...
                sum(error4eCRL2(c4n,n4e,@(x)uExact2(x),u(:,2))));
        % Energy exact error
        energy4lvl(end+1) = ...
            sqrt(sum(error4eCREnergy(c4n,n4e,@(x)gradExact1(x),u(:,1)))+...
                sum(error4eCREnergy(c4n,n4e,@(x)gradExact2(x),u(:,2))));
        % Stress exact error
        stress4lvl(end+1) = ...
                sqrt(sum(error4eStokesCRStress(c4n,n4e,1,...
                    @(x)sigmaExact1(x),u(:,1),p,gradU4e1))+...
                sum(error4eStokesCRStress(c4n,n4e,2,...
                    @(x)sigmaExact2(x),u(:,2),p,gradU4e2)));
        %% ESTIMATE
        [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,...
                    @(x)f(x),@(x)Du4Db1(x),@(x)Du4Db2(x),u,gradU4e);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4e));
        % print information on afem loop
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        % break condition
        if nrDoF >= minNrDoF, break, end;
        %% MARK
        if theta==1
            n4sMarked = markUniform(n4e);
        else
            n4sMarked = markBulk(n4e,eta4e,theta);
        end
        %% REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    %% Plot mesh, solution and convergence graph
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    % Mesh
    figure;
    plotTriangulation(c4n,n4e);
    % Solution
    figure;
    plotCRP0(c4n,n4e,u,p);
    % Convergence graphs
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u-u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nabla u - \nabla u_l||_{L2}');
    plotConvergence(nrDoF4lvl,stress4lvl,'||\sigma - \sigma_l||_{L2}');
    plotConvergence(nrDoF4lvl,averaging4lvl,'||\sigma_l - A \sigma_l||_{L2}');
```

```
end

%% PROBLEM INPUT DATA
function val = uExact1(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = 1.5*sqrt(r).*(cos(0.5*theta)-cos(1.5*theta));
end

function val = uExact2(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = 1.5*sqrt(r).*(3*sin(0.5*theta)-sin(1.5*theta));
end

function val = f(x)
val = zeros(size(x));
end

function val = u4Db(x)
val = [uExact1(x) uExact2(x)];
end

function val = g(x)
val = zeros(size(x));
end

function val = gradExact1(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = - sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du1Dr = 0.75*(r.^-0.5).*(cos(0.5*theta)-cos(1.5*theta));
Du1Dtheta = 0.75*sqrt(r).*(-sin(0.5*theta)+3*sin(1.5*theta));
val(:,1) = DrDx .* Du1Dr + DthetaDx .*Du1Dtheta;
val(:,2) = DrDy .* Du1Dr + DthetaDy .*Du1Dtheta;
end

function val = gradExact2(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
DrDx = cos(theta);
DthetaDx = -sin(theta)./r;
DrDy=sin(theta);
DthetaDy = cos(theta)./r;
Du2Dr = 0.75*(r.^-0.5).*(3*sin(0.5*theta)-sin(1.5*theta));
Du2Dtheta = 2.25*sqrt(r).*(cos(0.5*theta)-cos(1.5*theta));
val(:,1) = DrDx .* Du2Dr + DthetaDx .*Du2Dtheta;
val(:,2) = DrDy .* Du2Dr + DthetaDy .*Du2Dtheta;
end

function val = Du4Db1(x)
val = gradExact1(x);
```

```
end

function val = Du4Db2(x)
val = gradExact2(x);
end

function val = pExact(x)
[theta,r] = cart2polALT(x(:,1),x(:,2));
val = -6*(r.^-0.5).*cos(0.5*theta);
end

function val = sigmaExact1(x)
val = gradExact1(x) - [pExact(x), zeros(size(x,1),1)];
end

function val = sigmaExact2(x)
val=gradExact2(x) - [zeros(size(x,1),1), pExact(x)];
end

function [theta r] = cart2polALT(x,y)
% modified cart2pol for the angle convention
% theta in [0,2pi] instaed of [-pi,pi]
[theta,r] = cart2pol(x,y);
[ind] = theta<0;
theta(ind) = theta(ind)+2*pi;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.5: afemCRPoisson.m

```
function afemCRPoisson
%% afemCRPoisson - Solve Poisson model problem with CR elements.
% Solve the Poisson equation with linear CR finite elements adaptively on
% the domain Omega.
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N.

    %% Initialization
    addpath(genpath(pwd));
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry('Lshape',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);

    %% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveCRPoisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        % ESTIMATE
        [eta4s,n4s] = estimateCREtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    %% Plot mesh, solution and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotCR(c4n,n4e,x,{'CR Solution'; [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
end

%% problem input data
function val = f(x)
    val = ones(size(x,1),1);
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
```

```
end

function val = g(x)
    val = ones(size(x,1),1);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.6: afemCRPoissonLShapeExact.m

```
function afemCRPoissonLShapeExact
%% afemCRPoisson - Solve Poisson model problem with CR elements.
% Solve the Poisson equation with linear CR finite elements adaptively on
% the domain Omega.
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u (in polar coordinates):
% u = r^(2/3)*sin(2/3*phi)

%% Initialization
    addpath(genpath(pwd));
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry('LshapeNb',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);
    error4lvl = [];
    energy4lvl = [];
    nrDoF4lvl = [];
```

```
%% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveCRPoisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        %Exact error
        error4lvl(end+1) = sqrt(sum(error4eCRL2(c4n, n4e, @uExact, x)));
        %Energy error
        energy4lvl(end+1) = ...
           sqrt(sum(error4eCREnergy(c4n, n4e, @gradExact, x)));
        % ESTIMATE
        [eta4s,n4s] = estimateCREtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

%% Plot mesh, solution and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotCR(c4n,n4e,x,{'CR Solution'; [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u - u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nablau - \nablau_l||_{L2}');
end

%% problem input data
function val = f(x)
   val = zeros(size(x,1),1);
end

function val = u4Db(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    val = r.^(2/3).*sin(2/3*phi);
end

function val = g(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
```

```
    phi(index) = phi(index) + 2*pi;
    if ~isempty( find(phi<0)) && ~isempty(find(phi>2*pi))
        error('winkel')
    end

    for i = 1 : (size(x,1))
        if phi(i)>=0 && phi(i)<1/4*pi
            N = [1;0];
        elseif phi(i)>=1/4*pi && phi(i)<3/4*pi
            N = [0;1];
        elseif phi(i)>=3/4*pi && phi(i)<5/4*pi
            N = [-1;0];
        elseif phi(i)>=5/4*pi && phi(i)<6/4*pi
            N = [0;-1];
        else
            error('rand')
        end
        N = [-sin(phi(i)), cos(phi(i)); cos(phi(i)), sin(phi(i))]*N;
        val(i,:) = 2/3*r(i)^(-1/3)*[cos(2/3*phi(i)), sin(2/3*phi(i))]*N;
    end
end

function val = uExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    val = r.^(2/3).*sin(2/3*phi);
end

function val = gradExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    if ~isempty( find(phi<0)) && ~isempty(find(phi>2*pi))
        error('winkel')
    end

    val(:,1) = 2/3*r.^(-1/3).*cos(2/3*phi);
    val(:,2) = 2/3*r.^(-1/3).*sin(2/3*phi);
    for i = 1 : (size(x,1))
        if r(i)==0
            val(i,2)=Inf;
        end
        val(i,:)=val(i,:)*[-sin(phi(i)), cos(phi(i));...
            cos(phi(i)), sin(phi(i))];
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
```

```
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.7: afemCRPoissonSquareExact.m

```
function afemCRPoissonSquareExact
%% afemCRPoisson - Solve Poisson model problem with CR elements.
% Solve the Poisson equation with linear CR finite elements adaptively on
% the domain Omega.
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u:
% u = x(1-y)y(1-x)

%% Initialization
    addpath(genpath(pwd));
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry('SquareNb',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);
    error4lvl = [];
    energy4lvl = [];
    nrDoF4lvl = [];

%% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveCRPoisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
               %Exact error
        error4lvl(end+1) = sqrt(sum(error4eCRL2(c4n, n4e, @uExact, x)));
               %Energy error
        energy4lvl(end+1) = ...
```

```
            sqrt(sum(error4eCREnergy(c4n, n4e, @gradExact, x)));
        % ESTIMATE
        [eta4s,n4s] = estimateCREtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

%% Plot mesh, solution and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotCR(c4n,n4e,x,{'CR Solution'; [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'Eta');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'Lï¿½-Error');
    plotConvergence(nrDoF4lvl,energy4lvl,'Energy-Error');
end

%% problem input data
function val = f(x)
     val = 2*x(:,1) - 2*x(:,1).^2 + 2*x(:,2) - 2*x(:,2).^2;
end

function val = u4Db(x)
     val = zeros(size(x,1),1);
end

function val = g(x)
    x1 = x(:,1);
    x2 = x(:,2);
    if isempty(x)
        val = zeros(0,1);
    else
        for i = 1:(size(x,1))
            if x1(i)==0
                N = [-1;0];
            elseif x2(i)==0
                N = [0;-1];
            elseif x1(i)==1
                N = [1;0];
            elseif x2(i)==1
                N = [0;1];
            end
```

```
            val(i,:) = (x2(i) - x2(i)^2 - 2*x1(i)*x2(i) +...
                2*x1(i)*x2(i)^2)*N(1,1) + (x1(i)-x1(i)^2-2*x1(i)*x2(i) +...
                2*x2(i)*x1(i)^2)*N(2,1);
        end
    end
end

function val = uExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = x1.*(1-x2).*x2.*(1-x1);
end

function val = gradExact(x)
    x1=x(:,1);
    x2=x(:,2);
    val= [x2 - x2.^2 - 2*x1.*x2 + 2*x1.*x2.^2,...
              x1 - x1.^2 - 2*x1.*x2 + 2*x2.*x1.^2];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.8: afemIntegrate.m

```
function afemIntegrate

addpath(genpath(pwd));

[c4n, n4e, n4sDb, n4sNb] = loadGeometry('Square',1);

global p;
```

```
p = 10;

degree = 10;

disp(' ');
disp(['p: ',int2str(p),' degree: ',int2str(degree)]);

val = integrate(c4n,[n4sDb;n4sNb],@f1D,degree);
error_1D = norm(2+4/(p+1)-sum(val));

val = integrate(c4n,n4e,@f2D,degree);
error_2D = norm(2/(p+1)-sum(val));

disp(['Error 1D: ',num2str(error_1D),' Error 2D: ',num2str(error_2D)]);

val = integrate(c4n,n4e,@fMatrix,degree);
error_Matrix1 = norm(2/(p+1)-sum(val(:,1,1)));
error_Matrix2 = norm(2/(p+1)-sum(val(:,1,1)));

disp(['Error Matrix: ',num2str(error_Matrix1),...
      ' Error Matrix: ',num2str(error_Matrix2)]);

val = integrate(c4n,n4e,@fMatrix,degree);
error_Matrix1 = norm(2/(p+1)-sum(val(:,1,1)));
error_Matrix2 = norm(2/(p+1)-sum(val(:,1,1)));

disp(['Error Matrix: ',num2str(error_Matrix1),...
      ' Error Matrix: ',num2str(error_Matrix2)]);

l2norm = L2Norm(c4n,n4e,@f2D,2*degree);
error_L2Norm = norm(sqrt(2/(2*p+1)+2/(p+1)^2)-l2norm);

disp(['Error L2 Norm: ',num2str(error_L2Norm)]);

osc = oscillations(c4n,n4e,@fOsc,degree);
disp(['Oscillations: ',num2str(norm(osc))]);


function val = f1D(parts,c4parts,gp)
global p;
val = c4parts(:,1).^p+c4parts(:,2).^p;

function val = f2D(parts,c4parts,gp)
global p;
val = c4parts(:,1).^p+c4parts(:,2).^p;

function val = fMatrix(parts,c4parts,gp)
global p;
val(:,1,1) = c4parts(:,1).^p+c4parts(:,2).^p;
val(:,1,2) = c4parts(:,1).^p+c4parts(:,2).^p;

function val = fOsc(c4parts)
```

```
global p;
val = c4parts(:,1).^p+c4parts(:,2).^p;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.9: afemP1P1Elasticity.m

```
function afemP1P1Elasticity
% afemP1Poisson.m

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('Lshape',3);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);

    %problem dependent parameters
    E = 100000; %set by the user
    nu = 0.3; %set by the user
    mu = E/(2*(1+nu));
    lambda = E*nu/((1+nu)*(1-2*nu));

    %% AFEM loop
    [x, nrDoF] = ...
        solveP1P1Elasticity(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb,mu,lambda);
    %% Plot the solution
    plotP1P1(n4e,c4n,x,lambda,mu,20);
end
```

```
%% problem input data
function val = f(x)
    val = zeros(size(x,1),2);
end

function [W,M] = u4Db(x,lambda,mu)
    M = zeros(2*size(x,1),2);
    W = zeros(2*size(x,1),1);
    M(1:2:end,1) = 1;
    M(2:2:end,2) = 1;
    value = u_value(x,lambda,mu);
    W(1:2:end,1) = value(:,1);
    W(2:2:end,1) = value(:,2);
end

function val = g(x)
    val = zeros(size(x,1),1); %?!?! dim unklar
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.10: afemP1P1ElasticityCookMembrane.m

```
function afemP1P1ElasticityCookMembrane
% afemP1Poisson.m

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('CookMembrane',4);
% minNrDoF = 1000;
% eta4nrDoF = sparse(1,1);
```

```
    %problem dependent parameters
    E = 2900; %set by the user
    nu = 0.4; %set by the user
    mu = E/(2*(1+nu));
    lambda = E*nu/((1+nu)*(1-2*nu));

    %% AFEM loop
    [x, nrDoF] = ...
        solveP1P1Elasticity(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb,mu,lambda);
    %% Plot the solution
    plotP1P1(n4e,c4n,x,lambda,mu,20);
end

%% problem input data
function val = f(x)
    val = zeros(size(x,1),2);
end

function [W,M] = u4Db(x,lambda,mu)
    M = zeros(2*size(x,1),2);
    W = zeros(2*size(x,1),1);
    M(1:2:end,1) = 1;
    M(2:2:end,2) = 1;
% value = u_value(x,lambda,mu);
% W(1:2:end,1) = value(:,1);
% W(2:2:end,1) = value(:,2);
end

function val = g(x,n)
    val = zeros(size(x,1),2);
    val(find(n(:,1)==1),2)=1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.11: afemP1P1ElasticitySquare.m

```
function afemP1P1ElasticitySquare
% afemP1Poisson.m

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('Square',3);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);

    %problem dependent parameters
    E = 100000; %set by the user
    nu = 0.3; %set by the user
    mu = E/(2*(1+nu));
    lambda = E*nu/((1+nu)*(1-2*nu));

    %% AFEM loop
    [x, nrDoF] = ...
        solveP1P1Elasticity(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb,mu,lambda);
    %% Plot the solution
    plotP1P1(n4e,c4n,x,lambda,mu,20);
end

%% problem input data
function val = f(x)
    val = zeros(size(x,1),2);
end

function [W,M] = u4Db(x,lambda,mu)
    M = zeros(2*size(x,1),2);
    W = zeros(2*size(x,1),1);
    M(1:2:end,1) = 1;
    M(2:2:end,2) = 1;
    value = u_value(x,lambda,mu);
    W(1:2:end,1) = value(:,1);
    W(2:2:end,1) = value(:,2);
end

% function val = g(x)
% val = zeros(size(x,1),1); %?!?! dim unklar
% end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
```

```
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.12: afemP1P1ElasticitySquareExact.m

```
function afemP1P1Elasticity
% afemP1Poisson.m

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('SquareNb2',1);
    minNrDoF = 10000;
    eta4nrDoF = sparse(1,1);

    %problem dependent parameters
    E = 100000; %set by the user
    nu = 0.3; %set by the user
    mu = E/(2*(1+nu));
    lambda = E*nu/((1+nu)*(1-2*nu));

    %% AFEM loop
    [x, nrDoF] = ...
        solveP1P1Elasticity(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb,mu,lambda);
    %% Plot the solution
    plotP1P1(n4e,c4n,x,lambda,mu,20);
end

%% problem input data
function val = f(x)
    val = zeros(size(x,1),2);
end

function [W,M] = u4Db(x,lambda,mu)
    M = zeros(2*size(x,1),2);
    W = zeros(2*size(x,1),1);
    M(1:2:end,1) = 1;
```

```
    M(2:2:end,2) = 1;
    value = uExact(x,lambda,mu);
    W(1:2:end,1) = value(:,1);
    W(2:2:end,1) = value(:,2);
end

function val = g(x,n)
    %val = zeros(size(x,1),2);
    val = [2*cos(2 + 2*x(:,1)) .* cos(1 + x(:,2)).^2, -2*cos(1 + x(:,2)) .* sin(2 + 2*x(:,1)) .* sin(1 +
           -pi*sin(pi*(1 + x(:,1))) .* sin(1 + x(:,2)).^2, 2*cos(pi*(1 + x(:,1))) .* cos(1 + x(:,2)) .* s
    val = val * n;


    %{ { 2 Cos[2 (1 + x)] Cos[1 + y]^2,
    % -2 Cos[1 + y] Sin[2 (1 + x)] Sin[1 + y]},
    % { -\[Pi] Sin[\[Pi] (1 + x)] Sin[1 + y]^2,
    % 2 Cos[\[Pi] (1 + x)] Cos[1 + y] Sin[1 + y]
    %} }
end

function val = uExact(x, lambda, mu)
    val = [sin(2*(x(:,1) + 1)) .* cos(x(:,2) + 1).^2,...
           cos(pi*(x(:,1) + 1)) .* sin(x(:,2) + 1).^2];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.13: afemP1Poisson.m

```
function afemP1Poisson
% afemP1Poisson.m
```

```
% Solve the Poisson equation with linear P1 finite elements adaptively.
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N.

    %% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('Lshape',1);
    minNrDoF = 10000;
    eta4nrDoF = sparse(1,1);

    %% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        % ESTIMATE
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    %% Plot mesh, solution and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotP1(c4n,n4e,x,{'P1 Solution' [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
end

%% problem input data
function val = f(x)
    val = ones(size(x,1),1);
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    val = ones(size(x,1),1);
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.14: afemP1PoissonLShapeExact.m

```
function afemP1PoissonLShapeExact
% afemP1PoissonLShapeExact.m
% Solve the Poisson equation on an L-Shape domain with linear P1 finite
% elements adaptively.
%
% Given: function f. Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u (in polar coordinates):
% u = r^(2/3)*sin(2/3*phi)
%% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('LshapeNb',1);
    minNrDoF = 5000;
    eta4nrDoF = sparse(1,1);

%% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        %Exact error
        error4e = error4eP1L2(c4n,n4e,@uExact,x);
```

```
        error4nrDoF(nrDoF) = sqrt(sum(error4e));
               %Energy error
        en_error4e = error4eP1Energy(c4n,n4e,@gradExact,x);
        en_error4nrDoF(nrDoF) = sqrt(sum(en_error4e));
               % ESTIMATE
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

%% Output error
    disp(['L2-Norm of the exact error: ',num2str(error4nrDoF(nrDoF))]);
    disp(['L2-Norm of the energy error: ',num2str(en_error4nrDoF(nrDoF))]);

%% Plot solution and convergence graph.
    figure;
    subplot(1,2,1);
    plotP1(c4n,n4e,uExact(c4n),'Exact Solution');
    subplot(1,2,2);
    plotP1(c4n,n4e,x,'P1 Solution');
    figure;
    plotP04e(c4n,n4e,error4e,'L2 error on elements');
    figure;
    plotP1(c4n,n4e,abs(uExact(c4n)-x),...
        'Difference of exact and approximated solution');
    figure;
    plotP04e(c4n,n4e,en_error4e,...
        {'Exact error of the gradients on elements',...
        '(grad u - grad u_l)'});
    nrDoF4lvl = find(eta4nrDoF);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    error4lvl=error4nrDoF(nrDoF4lvl);
    en_error4lvl = en_error4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl);
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl);
    plotConvergence(nrDoF4lvl,en_error4lvl,'L2 energy error');

end

%% problem input data
function val = f(x)
   val = zeros(size(x,1),1);
end
```

```
function val = u4Db(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    val = r.^(2/3).*sin(2/3*phi);
end

function val = g(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    if ~isempty( find(phi<0)) && ~isempty(find(phi>2*pi))
        error('winkel')
    end
    for i = 1 : (size(x,1))
        if phi(i) >= 0 && phi(i) < 1/4*pi
            N = [1;0];
        elseif phi(i) >= 1/4*pi && phi(i) < 3/4*pi
            N = [0;1];
        elseif phi(i) >= 3/4*pi && phi(i) < 5/4*pi
            N = [-1;0];
        elseif phi(i) >= 5/4*pi && phi(i) < 6/4*pi
            N = [0;-1];
        else
            error('rand')
        end
        N=[-sin(phi(i)), cos(phi(i)); cos(phi(i)), sin(phi(i))]*N;
        val(i,:) = 2/3*r(i)^(-1/3)*[ cos(2/3*phi(i)), ...
            sin(2/3*phi(i)) ]*N;
    end
end

function val = uExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index) + 2*pi;
    val = r.^(2/3).*sin(2/3*phi);
end

function val = gradExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    [index] = find(phi<0);
    phi(index) = phi(index)+2*pi;
    if ~isempty( find(phi<0)) && ~isempty(find(phi>2*pi))
        error('winkel')
    end
    val(:,1) = 2/3*r.^(-1/3).*cos(2/3*phi);
    val(:,2) = 2/3*r.^(-1/3).*sin(2/3*phi);
    for i = 1 : (size(x,1))
        if r(i)==0
            val(i,2) = Inf;
        end
```

```
        val(i,:)=val(i,:)*[-sin(phi(i)), cos(phi(i));...
            cos(phi(i)), sin(phi(i))];
    end
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.15: afemP1PoissonShort.m

```
function afemP1PoissonShort

addpath(genpath(pwd));
[c4n, n4e, n4sDb, n4sNb] = loadGeometry('Square',1);
eta4nrDoF = sparse(1,1);

for l = 1 : 6
    % SOLVE
    [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
    % ESTIMATE
    [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
    eta4nrDoF(nrDoF) = norm(eta4s);
    % display
    disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
        '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
    figure;
    plotP1(c4n,n4e,x,{'P1-Solution' [num2str(nrDoF) ' degrees of freedom']});pause(0.1)
    % MARK
    n4sMarked = markBulk(n4s,eta4s);
```

```
    % REFINE
    [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);

end

figure;
plotConvergence(find(eta4nrDoF),nonzeros(eta4nrDoF)','\eta_l');
end

%% problem input data
function val = f(x)
    val = ones(size(x,1),1);
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    val = ones(size(x,1),1);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.16: afemP1PoissonSlitExact.m

```
function afemP1PoissonSlitExact
% afemRTOPoisson.m
% Solve the Poisson equation with RTO P0 finite elements adaptively on
% a given geometry
```

```
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u (in polar coordinates):
% u = r^(2/3)*sin(2/3*phi)

%% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('SlitNb',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);
    error4nrDoF=sparse(1,1);

%% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        %Exact error
        error4e = error4eP1L2(c4n,n4e,@uExact,x);
        error4nrDoF(nrDoF) = sqrt(sum(error4e));
        %Energy error
        energy4e = error4eP1Energy(c4n,n4e,@gradExact,x);
        energy4nrDoF(nrDoF) = sqrt(sum(energy4e));
        % ESTIMATE
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF)),'; L2-Fehler = ', num2str(error4nrDoF(nrD
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

%% Ouput error
    disp(['L2-Norm of the exact error: ',num2str(error4nrDoF(nrDoF))]);
    disp(['L2-Norm of the exact error for the gradient: ',...
        num2str(energy4nrDoF(nrDoF))]);

%% Plot mesh, solution, error and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotP1(c4n,n4e,x,{'P1 Solution' [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    error4lvl = error4nrDoF(nrDoF4lvl);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    energy4lvl = energy4nrDoF(nrDoF4lvl);
```

```
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u - u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nablau - \nablau_l||_{L2}');
end

%% problem input data
function val = f(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    % laplace u = d^2u/dr^2 + 1/r*du/dr + 1/r^2 * d^2u/dphi^2
    val = zeros(size(x,1),1);
end


function val = u4Db(x)
    val = uExact(x);
end


function val = g(x)
    [phi,r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    val = zeros(size(x,1),1);
    for i = 1 : (size(x,1))
        if (x(i,1)==1)
            N = [1;0];
        elseif (x(i,2)==1)
            N = [0;1];
        elseif (x(i,1)==-1)
            N = [-1;0];
        elseif (x(i,2)==-1)
            N = [0;-1];
        elseif (x(i,2)==0)
            N = [0;-1];
        else
            error('Normalen am Neumannrand');
        end
        val(i) = gradExact(x) * N;
    end
end


function val = uExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    val = r.^(1/4).*sin(1/4*phi);
end


function val = gradExact(x)
```

```
    x1 = x(:,1);
    x2 = x(:,2);
    [phi, r] = cart2pol(x1,x2);
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    if (~isempty(phi(phi<-eps)) | ~isempty(phi(phi>2*pi)))
        error('umrechnung in polarkoordinaten')
    end
    val(:,1) = 1/4*r.^(-3/4).*cos(1/4*phi);
    val(:,2) = 1/4*r.^(-3/4).*sin(1/4*phi);
    for i = 1 : (size(x,1))
        if r(i)==0
            val(i,2) = Inf;
        end
        val(i,:)=val(i,:)*[-sin(phi(i)), cos(phi(i));...
            cos(phi(i)), sin(phi(i))];
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.17: afemP1PoissonSquareExact.m

```
function afemP1PoissonSquareExact
% afemP1PoissonSquareExact.m
% Solve the Poisson equation with linear P1 finite elements adaptively.
%
% Given: function f. Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
```

```
% discrete solution with the exact solution u:
% u = x(1-y)y(1-x)
%% Initialization
    addpath(genpath(pwd));
    [c4n n4e n4sDb n4sNb] = loadGeometry('SquareNb',1);
    minNrDoF = 1000;
    eta4nrDoF = sparse(1,1);
    error4nrDoF = sparse(1,1);
    energy4nrDoF = sparse(1,1);
%% AFEM loop
    while( true )
        % SOLVE
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        %Exact error
        error4e = error4eP1L2(c4n,n4e,@uExact,x);
        error4nrDoF(nrDoF) = sqrt(sum(error4e));
        %Energy error
        energy4e = error4eP1Energy(c4n,n4e,@gradExact,x);
        energy4nrDoF(nrDoF) = sqrt(sum(energy4e));
        % ESTIMATE
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
            eta4nrDoF(nrDoF) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
            '; estimator = ',num2str(eta4nrDoF(nrDoF))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

%% Ouput error
    disp(['L2-Norm of the exact error: ',num2str(error4nrDoF(nrDoF))]);
    disp(['L2-Norm of the exact error for the gradient: ',...
        num2str(energy4nrDoF(nrDoF))]);

%% Plot mesh, solution, error and convergence graph.
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotP1(c4n,n4e,x,{'P1 Solution' [num2str(nrDoF) ' degrees of freedom']});
    nrDoF4lvl = find(eta4nrDoF);
    error4lvl = error4nrDoF(nrDoF4lvl);
    eta4lvl = eta4nrDoF(nrDoF4lvl);
    energy4lvl = energy4nrDoF(nrDoF4lvl);
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u - u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nablau - \nablau_l||_{L2}');
end
```

```
%% problem input data
function val = f(x)
    val = 2*x(:,1) - 2*x(:,1).^2 + 2*x(:,2) - 2*x(:,2).^2;
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    x1 = x(:,1);
    x2 = x(:,2);
    for i=1:(size(x,1))
        if x1(i)==0
            N = [-1;0];
        elseif x2(i)==0
            N = [0;-1];
        elseif x1(i)==1
            N = [1;0];
        elseif x2(i)==1
            N = [0;1];
        end
        val(i,:) = (x2(i) - x2(i)^2 - 2*x1(i)*x2(i) +...
            2*x1(i)*x2(i)^2)*N(1,1) + (x1(i) - x1(i)^2 - 2*x1(i)*x2(i) +...
            2*x2(i)*x1(i)^2)*N(2,1);
    end
end

function val = uExact(x)
    x1=x(:,1);
    x2=x(:,2);
    val = x1.*(1-x2).*x2.*(1-x1);
end

function val = gradExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = [x2 - x2.^2 - 2*x1.*x2 + 2*x1.*x2.^2,...
        x1 - x1.^2 - 2*x1.*x2 + 2*x2.*x1.^2];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
```

```
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.18: afemP1PoissonTeach.m

```
function afemP1PoissonTeach
% afemP1PoissonTeach.m
% Solve the Poisson equation with linear P1 finite elements adaptively on the.
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N.
%
% In addition the steps are visulized by 5 plots per level: the
% triangulation, the error estimates, the marked sides, the marked sides
% after the closure algorithm and the colored triangles (RGB)

    %% add paths
    addpath(genpath(pwd));

    %% load the geometry
    % geom = 'Square';
    geom = 'Lshape';
    % geom = 'Slit';
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry(geom,1);

    %% set the maximal number of nodes
    minNrDoFs = 1000;

    %% initialisation
    nrDoF4lvl = [];
    eta4lvl = [];

    %% AFEM loop
    % Solve and estimate at least once. Decide whether or not to continue
    % the AFEM loop afterwards.
    tic

    % solve
    [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
```

```
    nrDoF4lvl(end+1) = nrDoF;
    % plot first triangulation
    aFemLoopFigure = figure;
    set(aFemLoopFigure,'Name','AFEM-Teach','Position',[50 500 600 400]);
    angle = [-28, 56];
    [triX,triY,triZ] = getTriangulationXYZ(c4n, n4e);
    patch(triX,triY,[1 1 1]); %white triangles
    myaxis = axis;

    % estimate
    [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
    eta4lvl(end+1) = norm(eta4s);
    disp(['nodes/dofs: ',int2str(size(c4n,1)),'/',num2str(nrDoF),...
          '; estimator = ',num2str(eta4lvl(end))]);
    % plot first error estimates
    aFemErrorFigure = figure;
    plotEtaSidesTeach(aFemErrorFigure, c4n, n4s, eta4s);

    % While we have not reached the desired number of degrees of freedom
    % yet, execute the AFEM loop.
    while( nrDoF < minNrDoFs )

        % mark
        n4sMarked = markBulk(n4s,eta4s);
        % plot marked sides
        waitForUser();
        figure(aFemLoopFigure);
        set(0,'CurrentFigure',aFemLoopFigure);
        clf;
        [triX,triY,triZ] = getTriangulationXYZ(c4n, n4e);
        patch(triX,triY,triZ,[1 1 1]);
        [markX,markY,markC] = getMarkXYC(c4n, n4sMarked);
        axis(myaxis);
        patch(markX,markY,markC,'EdgeColor','none');

        % plot reference sides
        waitForUser();
        figure(aFemLoopFigure);
        set(0,'CurrentFigure',aFemLoopFigure);
        [refX,refY,refC] = getRefXYC(c4n, n4e);
        axis(myaxis);
        patch(refX,refY,refC);

        % plot marked sides after closure --> new edges with other color
        n4sRefine = closure(n4e,n4sMarked); % just for the plot!
        waitForUser();
        figure(aFemLoopFigure);
        set(0,'CurrentFigure',aFemLoopFigure);
        [markX,markY,markC] = getMarkXYC(c4n, n4sRefine);
        axis(myaxis);
        markC(1,:,:) = markC(1,:,[2 3 1]); % changes color to blue
        [n4sClosure, indexN4sRefine, indexN4sMarked] = intersect(n4sRefine, [n4sMarked; n4sMarke
```

```
        markC(1,indexN4sRefine,:) = ones(length(indexN4sRefine),1)*[1 0 0]; % already marked sides red ag
        patch(markX,markY,markC,'EdgeColor','none');

        % plot how afem will refine
        waitForUser();
        figure(aFemLoopFigure);
        set(0,'CurrentFigure',aFemLoopFigure);
        clf;
        [colX,colY,colC] = getColoredXYC(c4n, n4e, n4sRefine);
        patch(colX,colY,colC);
        %refine
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);

        % plot new triangulation
        waitForUser();
        figure(aFemLoopFigure);
        set(0,'CurrentFigure',aFemLoopFigure);
        clf;
        [triX,triY,triZ] = getTriangulationXYZ(c4n, n4e);
        patch(triX,triY,triZ,[1,1,1]);

        % solve
        [x,nrDoF] = solveP1Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        % estimate
        [eta4s,n4s] = estimateP1EtaSides(@f,@g,@u4Db,x,c4n,n4e,n4sDb,n4sNb);
        eta4lvl(end+1) = norm(eta4s);
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
             '; estimator = ',num2str(eta4lvl(end))]);
        % plot estimated error
        figure(aFemErrorFigure);
        set(0,'CurrentFigure',aFemErrorFigure);
        plotEtaSidesTeach(aFemErrorFigure, c4n, n4s, eta4s);
    end

    toc

    figure;
    plotP1(c4n,n4e,x,{'P1-Solution' [num2str(length(x)) ' nodes']});
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'Eta');
end

%% problem input data
function val = f(x)
    val = ones(size(x,1),1);
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end
```

```
function val = g(x)
    val = zeros(size(x,1),1);
end

%% functions for teach-plot
% function which returns the input parameters of the patch function (no Color) to
% draw a triangulation - also used as basis for the marking
function [valX, valY, valZ] = getTriangulationXYZ(c4n, n4e)
    % coordinates for triangles
    X1 = c4n(n4e(:,1), 1);
    X2 = c4n(n4e(:,2), 1);
    X3 = c4n(n4e(:,3), 1);
    Y1 = c4n(n4e(:,1), 2);
    Y2 = c4n(n4e(:,2), 2);
    Y3 = c4n(n4e(:,3), 2);

    % combined coordinates in patch-style
    valX = [X1';X2';X3'];
    valY = [Y1';Y2';Y3'];
    % no heigth
    valZ = zeros(size(valX));
end

% function which returns the input parameters of the patch function to
% mark the reference sides with a second parallel line.
% reference sides are the side between the first two nodes in n4e
function [valX, valY, valC] = getRefXYC(c4n, n4e)
    % coordinates of reference sides
    X1 = c4n(n4e(:,1), 1);
    X2 = c4n(n4e(:,2), 1);
    Y1 = c4n(n4e(:,1), 2);
    Y2 = c4n(n4e(:,2), 2);

    % combined coordinates in patch-style
    valX = [X1';X2'];
    valY = [Y1';Y2'];

    for i=1 : size(valX,2)
        v = [valX(1,i) - valX(2,i); valY(1,i) - valY(2,i)]; % direction-vector
        w = [-v(2,1);v(1,1)] / (norm(v)*90); % v*w = 0
        v = v / 4;

        % add or substract w and v to change the position of the line
        valX(1,i) = valX(1,i) - w(1,1) - v(1,1);
        valX(2,i) = valX(2,i) - w(1,1) + v(1,1);

        valY(1,i) = valY(1,i) - w(2,1) - v(2,1);
        valY(2,i) = valY(2,i) - w(2,1) + v(2,1);
    end

    % no color
    valC = zeros(size(valX));
```

```
194 end

    % function which returns the input parameters of the patch function to
    % draw red bold lines for marked sides
    function [valX, valY, valC] = getMarkXYC(c4n, n4sM)
199     % coordinates for all nodes of marked sides
        X1 = c4n(n4sM(:,1), 1);
        X2 = c4n(n4sM(:,2), 1);
        Y1 = c4n(n4sM(:,1), 2);
        Y2 = c4n(n4sM(:,2), 2);
204
        % combined coordinates in patch-style (would be lines up to here)
        X = [X1';X2'];
        Y = [Y1';Y2'];
        valX = zeros(4,size(X,2));
209     valY = zeros(4,size(Y,2));

        % makes boxes out of the edges so they can be seen better
        for i=1 : size(X,2)
            v = [X(1,i) - X(2,i); Y(1,i) - Y(2,i)]; % direction-vector
214         w = [-v(2,1);v(1,1)] / (norm(v)*100); % v*w = 0

            % add or substract w to make it bold
            valX(1,i) = X(1,i) - w(1,1);
            valX(2,i) = X(2,i) - w(1,1);
219         valX(3,i) = X(2,i) + w(1,1);
            valX(4,i) = X(1,i) + w(1,1);

            valY(1,i) = Y(1,i) - w(2,1);
            valY(2,i) = Y(2,i) - w(2,1);
224         valY(3,i) = Y(2,i) + w(2,1);
            valY(4,i) = Y(1,i) + w(2,1);
        end

        %color: red
229     valC = zeros(1,size(valX,2),3);
        valC(1,:,1) = ones(1,size(valX,2),1);
    end

    % function which returns the input parameters of the patch function to
234 % draw RGB-colored triangles (number of marked sides)
    function [valX, valY, valC] = getColoredXYC(c4n, n4e, n4sR)
        % coordinates for triangles
        X1 = c4n(n4e(:,1), 1);
        X2 = c4n(n4e(:,2), 1);
239     X3 = c4n(n4e(:,3), 1);
        Y1 = c4n(n4e(:,1), 2);
        Y2 = c4n(n4e(:,2), 2);
        Y3 = c4n(n4e(:,3), 2);

244     % combined coordinates in patch-style
        valX = [X1';X2';X3'];
```

```
    valY = [Y1';Y2';Y3'];

    % numbers of Nodes and Elements
    nrNodes = size(c4n,1);
    nrElems = size(n4e,1);
    valC = ones(1,nrElems,3); %default color: white

    % compute newNodes4n to find new nodes faster as in refineRGB.m
    newNode4n = sparse(n4sR(:,1),n4sR(:,2),(1:size(n4sR,1))'+ nrNodes, nrNodes, nrNodes);
    newNode4n = newNode4n + newNode4n';

    for curElem = 1 : nrElems
        % the three nodes of the current element
        curNodes = n4e(curElem,:);
        curNewNodes = [newNode4n(curNodes(1),curNodes(2));
                       newNode4n(curNodes(2),curNodes(3));
                       newNode4n(curNodes(3),curNodes(1));
                      ];
        if nnz(curNewNodes) == 1 % green if one side is marked
            valC(1,curElem,:) = [0 1 0];
        elseif nnz(curNewNodes) == 2 % blue if two sides are marked
            valC(1,curElem,:) = [0 0 1];
        elseif nnz(curNewNodes) == 3 % red if all sides are marked
            valC(1,curElem,:) = [1 0 0];
        end
    end
end

function waitForUser()
    input(sprintf('Enter to continue: '));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
```

```
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.19: afemRT0Poisson.m

```
function afemRT0Poisson
% afemRT0Poisson.m
% Solve the Poisson equation with RT0 P0 finite elements adaptively on
% a given geometry
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N.

    %% add paths
    addpath(genpath(pwd));

    close all;
    %% load the geometry
    %geom = 'Square';
    geom = 'Lshape';
    %geom = 'Slit';
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry(geom,1);

    %% set the minimal number of nodes
    minNrDoF = 1000;

    %% initialisation
    nrDoF4lvl = [];
    eta4lvl = [];

    %% AFEM loop
    tic
    while( true )
        % SOLVE
        [p,u,nrDoF] = solveRT0Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        % ESTIMATE
        [eta4s,n4s] = estimateRT0EtaSides(@f,@g,@u4Db,p,u,c4n,n4e,n4sDb,n4sNb);
        eta4lvl(end+1) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
              '; estimator = ',num2str(eta4lvl(end))]);
        % leave the loop if minNrDoF is reached
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end
```

```
    toc

    %% plot
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotP04e(c4n,n4e,u,...
       {'RT0 Solution - u'...
       [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotRT04e(c4n,n4e,p,{'RT0 Solution - p'...
        [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
end

%% problem input data
function val = f(x)
    val = ones(size(x,1),1);
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    val = zeros(size(x,1),1);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
```

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

Listing 3.20: afemRT0PoissonSlitExact.m

```
function afemRT0PoissonSlitExact
% afemRT0Poisson.m
% Solve the Poisson equation with RT0 P0 finite elements adaptively on
% a given geometry
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u (in polar coordinates):
% u = r^(1/4)*sin(1/4*phi)

%% Initialization
    addpath(genpath(pwd));
    close all;
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry('SlitNb',1);
    minNrDoF = 1000;
    nrDoF4lvl = [];
    eta4lvl = [];
    error4lvl = [];
    energy4lvl = [];

%% AFEM loop
    tic
    while( true )
        % SOLVE
        [p,u,nrDoF] = solveRT0Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        %Exact error
        error4e = error4eRT0L2(c4n, n4e, @uExact, u);
        error4lvl(end+1) = sqrt(sum(error4e));
        %Energy error
        energy4e = error4eRT0Energy(c4n, n4e, @gradExact, p);
        energy4lvl(end+1) = sqrt(sum(energy4e));
        % ESTIMATE
        [eta4s,n4s] = estimateRT0EtaSides(@f,@g,@u4Db,p,u,c4n,n4e,n4sDb,n4sNb);
        eta4lvl(end+1) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
              '; estimator = ',num2str(eta4lvl(end))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end

    toc
```

```
    %% plot
    figure;
    plotTriangulation(c4n,n4e);
    figure;
    plotP04e(c4n,n4e,u,...
       {'RT0 Solution - u'...
       [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotRT04e(c4n,n4e,p,{'RT0 Solution - p'...
        [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'||u - u_l||_{L2}');
    plotConvergence(nrDoF4lvl,energy4lvl,'||\nablau - \nablau_l||_{L2}');
end

%% problem input data
function val = f(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    % laplace u = d^2u/dr^2 + 1/r*du/dr + 1/r^2 * d^2u/dphi^2
    val = zeros(size(x,1),1);
end


function val = u4Db(x)
    val = uExact(x);
end


function val = g(x)
    [phi,r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    val = zeros(size(x,1),1);
    for i = 1 : (size(x,1))
        if (x(i,1)==1)
            N = [1;0];
        elseif (x(i,2)==1)
            N = [0;1];
        elseif (x(i,1)==-1)
            N = [-1;0];
        elseif (x(i,2)==-1)
            N = [0;-1];
        elseif (x(i,2)==0)
            N = [0;-1];
        else
            error('Normalen am Neumannrand');
        end
        val(i) = gradExact(x) * N;
    end
end
```

```
function val = uExact(x)
    [phi, r] = cart2pol(x(:,1),x(:,2));
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    val = r.^(1/4).*sin(1/4*phi);
end


function val = gradExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    [phi, r] = cart2pol(x1,x2);
    phi(phi < -eps) = phi(phi < -eps)+2*pi;
    if (~isempty(phi(phi<-eps)) | ~isempty(phi(phi>2*pi)))
        error('umrechnung in polarkoordinaten')
    end
    val(:,1) = 1/4*r.^(-3/4).*cos(1/4*phi);
    val(:,2) = 1/4*r.^(-3/4).*sin(1/4*phi);
    for i = 1 : (size(x,1))
        if r(i)==0
            val(i,2) = Inf;
        end
        val(i,:)=val(i,:)*[-sin(phi(i)), cos(phi(i));...
            cos(phi(i)), sin(phi(i))];
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.21: afemRT0PoissonSquareExact.m

```
function afemRT0PoissonSquareExact
% afemRT0Poisson.m
% Solve the Poisson equation with RT0 P0 finite elements adaptively on
% a given geometry
%
% Seek for a solution u such that
% -div(grad(u)) = f in Omega,
% u = 0 on Gamma_D,
% u*n = g on Gamma_N.
% with Dirichlet boundary Gamma_D and Neumann boundary Gamma_N. Compare the
% discrete solution with the exact solution u:
% u = x(1-y)y(1-x)

%% Initialization
    addpath(genpath(pwd));
    close all;
    [c4n, n4e, n4sDb, n4sNb] = loadGeometry('Square',1);
    minNrDoF = 10000;
    nrDoF4lvl = [];
    eta4lvl = [];
    error4lvl = [];
    energy4lvl = [];

%% AFEM loop
    tic
    while( true )
        % SOLVE
        [p,u,nrDoF] = solveRT0Poisson(@f,@g,@u4Db,c4n,n4e,n4sDb,n4sNb);
        nrDoF4lvl(end+1) = nrDoF;
        % L2 Error
        error4lvl(end+1) = sqrt(sum(error4eRT0L2(c4n, n4e, @uExact, u)));
        % Energy Error
        energy4lvl(end+1) = ...
            sqrt(sum(error4eRT0Energy(c4n, n4e, @gradExact, p)));
        % ESTIMATE
        [eta4s,n4s] = estimateRT0EtaSides(@f,@g,@u4Db,p,u,c4n,n4e,n4sDb,n4sNb);
        eta4lvl(end+1) = sqrt(sum(eta4s));
        disp(['nodes/dofs: ',num2str(size(c4n,1)),'/',num2str(nrDoF),...
              '; estimator = ',num2str(eta4lvl(end))]);
        if nrDoF >= minNrDoF, break, end;
        % MARK
        n4sMarked = markBulk(n4s,eta4s);
        % REFINE
        [c4n,n4e,n4sDb,n4sNb] = refineRGB(c4n,n4e,n4sDb,n4sNb,n4sMarked);
    end
    toc

    %% plot
    figure;
    plotTriangulation(c4n,n4e);
    figure;
```

```
    plotP04e(c4n,n4e,u,...
       {'RT0 Solution - u'...
       [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotRT04e(c4n,n4e,p,{'RT0 Solution - p'...
        [num2str(length(p) + length(u)) ' degrees of freedom']});
    figure;
    plotConvergence(nrDoF4lvl,eta4lvl,'\eta_l');
    hold all;
    plotConvergence(nrDoF4lvl,error4lvl,'L^2-Error');
    plotConvergence(nrDoF4lvl,energy4lvl,'Energy-Error');
end

%% problem input data
function val = f(x)
    val = 2*x(:,1) - 2*x(:,1).^2 + 2*x(:,2) - 2*x(:,2).^2;
end

function val = u4Db(x)
    val = zeros(size(x,1),1);
end

function val = g(x)
    x1 = x(:,1);
    x2 = x(:,2);
    for i=1:(size(x,1))
        if x1(i)==0
            N=[-1;0];
        elseif x2(i)==0
            N=[0;-1];
        elseif x1(i)==1
            N=[1;0];
        elseif x2(i)==1
            N=[0;1];
        end
        val(i,:) = (x2(i)-x2(i)^2-2*x1(i)*x2(i) + 2*x1(i)*x2(i)^2)*N(1,1) +...
               (x1(i) - x1(i)^2 - 2*x1(i)*x2(i) + 2*x2(i)*x1(i)^2)*N(2,1);
    end
end

function val = uExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = x1.*(1-x2).*x2.*(1-x1);
end

function val = gradExact(x)
    x1 = x(:,1);
    x2 = x(:,2);
    val = [x2 - x2.^2 - 2*x1.*x2 + 2*x1.*x2.^2,...
          x1 - x1.^2 - 2*x1.*x2 + 2*x2.*x1.^2];
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.22: common/computeArea4e.m

```
function area4e = computeArea4e(c4n,n4e)
%% computeArea4e - Area for elements.
% computeArea4e(c4n, n4e) computes the area of each element of a
% decomposition where c4n, n4e are as specified in
% the documentation.
%
% See also: computeArea4n

    if isempty(n4e)
        area4e = zeros(0,1);
        return;
    end

    %% Compute area4e.
    % Get the x- and y-coordinates for each node of each element and
    % compute the area of all elements simulateously.
    x1 = c4n(n4e(:,1),1);
    x2 = c4n(n4e(:,2),1);
    x3 = c4n(n4e(:,3),1);
    y1 = c4n(n4e(:,1),2);
    y2 = c4n(n4e(:,2),2);
    y3 = c4n(n4e(:,3),2);

    area4e = ( x1.*(y2 - y3) + x2.*(y3 - y1) + x3.*(y1 - y2) )/2;
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.23: common/computeArea4n.m

```
function area4n = computeArea4n(c4n,n4e)
%% computeArea4n - Area for node patches.
% A node patch is the union of all elements a node is part of.
% computeArea4n(c4n, n4e) computes the area of the node patch for each node
% of a decomposition where c4n, n4e are as specified
% in the documentation.
%
% See also: computeArea4e

    if isempty(n4e)
        area4n = zeros(0,1);
        return;
    end

    %% Compute area4e.
    % Get the x- and y-coordinates for each node of each element and
    % compute the area of all elements simulateously.
    x1 = c4n(n4e(:,1),1);
    x2 = c4n(n4e(:,2),1);
    x3 = c4n(n4e(:,3),1);
    y1 = c4n(n4e(:,1),2);
    y2 = c4n(n4e(:,2),2);
    y3 = c4n(n4e(:,3),2);

```

```
    area4e = 1/2 * (x1.*(y2 - y3) + x2.*(y3 - y1) + x3.*(y1 - y2));

    %% Compute area4n.
    % Use area4n and n4e to accumulate area4n.
    area4e = area4e * ones(1,3);
    nrNodes = size(c4n,1);
    area4n = accumarray(n4e(:),area4e(:),[nrNodes 1]);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.24: common/computeE4n.m

```
function e4n = computeE4n(n4e)
%% computeE4n - Elements for nodes.
% computeE4n(n4e) returns a sparse matrix in which the entry (j,k) contains
% the number of the element that has the nodes j and k in
% counterclockwise order (or 0 if no such element exists).
% n4e is as specified in the documentation.
%
% See also: computeE4s

    if isempty(n4e)
        e4n = [];
        return;
    end

    %% Compute e4n.
    % Create a list of all sides in the decomposition and build a sparse
    % matrix such that each side computes its proper element number.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
```

```
    nrElems = size(n4e,1);
    N = max(max(n4e));
    elemNumbers = [1:nrElems 1:nrElems 1:nrElems];
    e4n = sparse(allSides(:,1),allSides(:,2),elemNumbers,N,N);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.25: common/computeE4s.m

```
function e4s = computeE4s(n4e)
%% computeE4s - Elements for sides.
% computeE4s(n4e) returns a matrix each row of which corresponds to one side
% of the decomposition. The side numbering is the same as in
% n4s. Each row contains the numbers of the two elements that
% the corresponding side is a part of. If it is a boundary
% side the second entry is 0.
% n4e is as specified in the documentation.
%
% See also: computeN4s

    if isempty(n4e)
        e4s = zeros(0,2);
        return;
    end

    %% Compute e4s.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
    [b,ind,back] = unique(sort(allSides,2),'rows','first');
    n4sInd = sort(ind); % by the way: n4s = allSides(n4sInd,:)
```

```
    nrElems = size(n4e,1);
    elemNumbers = [1:nrElems 1:nrElems 1:nrElems];
    e4s(:,1) = elemNumbers(n4sInd);
    allElem4s(ind) = accumarray(back,elemNumbers);
    e4s(:,2) = allElem4s(n4sInd)'-e4s(:,1);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.26: common/computeLength4s.m

```
function length4s = computeLength4s(c4n,n4s)
%% computeLength4s - Lengths for sides.
% computeLength4s(c4n, n4s) computes the length of each side of the
% decomposition. c4n and n4s are as specified in the
% documentation.
%
% See also: computeN4s, computeArea4e

    if isempty(n4s)
        length4s = zeros(0,1);
        return;
    end

    %% Compute length4s in a vectorised manner.
    length4s = sqrt( sum( (c4n(n4s(:,2),:) - c4n(n4s(:,1),:)).^2, 2) );
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
```

```
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.27: common/computeMid4e.m

```
function mid4e = computeMid4e(c4n, n4e)
%% computeMid4e - midpoints for elements.
% computeMid4e(c4n, n4e) computes the midpoint for each element of the
% decomposition. c4n and n4e are as specified in the
% documentation.
%
% See also: computeArea4e, computeMid4s

    if isempty(n4e)
        mid4e = zeros(0,2);
        return;
    end

    %% Compute mp4e.
    mid4e = ( c4n(n4e(:,1),:) + c4n(n4e(:,2),:) + c4n(n4e(:,3),:) ) / 3;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
```

```
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.28: common/computeMid4s.m

```
function mid4s = computeMid4s(c4n, n4s)
%% computeMid4s - midpoints for sides.
% computeMid4s(c4n, n4s) computes the midpoint for each side of the
% decomposition. c4n and n4s are as specified in the
% documentation.
%
% See also: computeN4s, computeMid4e

    if isempty(n4s)
        mid4s = zeros(0,2);
        return;
    end

    %% Compute mid4s.
    mid4s = 0.5 * ( c4n(n4s(:,1),:) + c4n(n4s(:,2),:) );
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.29: common/computeN4s.m

```
function n4s = computeN4s(n4e)
%% computeN4s - Nodes for sides.
% computeN4s(n4e) returns a matrix in which each row corresponds to one side
% of the decomposition. The side numbering is the same as in
% e4s, s4n, s4e, length4s, mp4s, normal4s and tangent4s. Each
% row consists of the numbers of the end nodes of the
% corresponding side. n4e is as specified in the
% documentation.
%
% See also: computeE4s, computeS4n, computeS4e, computeLength4s,
% computeMid4s, computeNormal4s, computeTangent4s

    if isempty(n4e)
        n4s = [];
        return;
    end

    %% Compute n4s.
    % Gather a list of all sides including duplicates (occurring for inner
    % sides), then sort each row and make sure the rows are unique, thus
    % eliminating duplicates.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
    % Eliminate duplicates, remember original index of remaining rows.
    [b,ind] = unique(sort(allSides,2),'rows','first');
    n4s = allSides(sort(ind),:);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.30: common/computeNormal4e.m

```
function normal4e = computeNormal4e(c4n,n4e)
%% computeNormal4e - normals for elements.
% computeNormal4e(c4n, n4e) computes the three outer unit normal vectors of
% each element of the decomposition. c4n and n4e
% are as specified in the documentation.
%
% See also: computeNormal4s, computeTangent4e

    if isempty(n4e)
        normal4e = zeros(0,2);
        return;
    end

    %% Compute normal4e.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
    c4start = c4n(allSides(:,1),:);
    c4end = c4n(allSides(:,2),:);
    lengths = sqrt(sum((c4end-c4start).^2,2));
    tangents = (c4end - c4start)./[lengths lengths];
    normals = [tangents(:,2), -tangents(:,1)];
    normal4e(1,:,:) = normals(1:size(n4e,1),:)';
    normal4e(2,:,:) = normals(size(n4e,1)+1:2*size(n4e,1),:)';
    normal4e(3,:,:) = normals(2*size(n4e,1)+1:3*size(n4e,1),:)';
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.31: common/computeNormal4s.m

```
function normal4s = computeNormal4s(c4n,n4s)
```

```
%% computeNormal4s - Normals for sides.
% computeNormal4s(c4n, n4s) computes the outer unit normal vectors for each
% side of the decomposition. For inner sides only
% one of the two possible normals is computed.
% c4n and n4s are as specified in the
% documentation.
%
% See also: computeN4s, computeNormal4e, computeTangent4s

    if isempty(n4s)
        normal4s = zeros(0,2);
        return;
    end

    %% Compute length4s.
    c4start = c4n(n4s(:,1),:);
    c4end = c4n(n4s(:,2),:);
    length4s = sqrt(sum((c4end-c4start).^2,2));

    %% Compute tangent4s.
    tangent4s = (c4end-c4start)./[length4s length4s];

    %% Compute normal4s.
    normal4s = [tangent4s(:,2), -tangent4s(:,1)];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.32: common/computeS4e.m

```
function s4e = computeS4e(n4e)
```

```
%% computeS4e - Sides for elements.
% computeS4e(n4e) returns a matrix each row of which corresponds to one
% element of the decomposition. Each row contains the numbers
% of the three sides belonging to an element. The side
% numbering is the same as in n4s.
%
% See also: computeN4s

    if isempty(n4e)
        s4e = [];
        return;
    end

    %% Compute s4e.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
    [b,ind,back] = unique(sort(allSides,2),'rows','first');
    [n4sInd, sortInd] = sort(ind); % by the way: n4s = allSides(n4sInd,:)
    sideNr(sortInd) = 1:length(ind); % sideNr(back): numbers for allSides
    s4e = reshape(sideNr(back),size(n4e));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.33: common/computeS4n.m

```
function s4n = computeS4n(n4e, n4s)
%% computeS4n - Sides for nodes.
% computeS4n(n4e) returns a symmetric sparse matrix in which the entry (j,k)
% contains the number of the side with the end nodes j and k
% or zero if no such side exists.
% The side numbering is the same as in n4s. n4e is as
```

```
% specified in the documentation.
%
% See also: computeN4s, computeS4e

  if isempty(n4e)
      s4n = [];
      return;
  end

  %% Optionally compute n4s.
  if nargin < 2
      n4s = computeN4s(n4e);
  end

  %% Compute s4n.
  S = size(n4s, 1);
  N = max(n4e(:));
  s4n = sparse(n4s(:, 1), n4s(:, 2), 1:S, N, N);
  % Up to here, s4n is not yet symmetric as each side has only
  % been considered once. The following makes sure that s4n is
  % symmetric.
  s4n = s4n + s4n';

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.34: common/computeTangent4e.m

```
function tangent4e = computeTangent4e(c4n,n4e)
```

```
%% computeTangent4e - Tangents for elements.
% computeTangent4e(c4n, n4e) computes the tangent vectors of all sides of all
% elements in the decomposition. The tangent
% vectors are normed and point in counterclockwise
% direction. c4n and n4e are as specified in the
% documentation.
%
% See also: computeTangent4s, computeNormal4e

    if isempty(n4e)
        tangent4e = zeros(0,2);
        return;
    end

    %% Compute tangent4e.
    allSides = [n4e(:,[1 2]); n4e(:,[2 3]); n4e(:,[3 1])];
    c4start = c4n(allSides(:,1),:);
    c4end = c4n(allSides(:,2),:);
    lengths = sqrt(sum((c4end-c4start).^2,2));
    tangents = (c4end - c4start)./[lengths lengths];
    tangent4e(1,:,:) = tangents(1:size(n4e,1),:)';
    tangent4e(2,:,:) = tangents(size(n4e,1)+1:2*size(n4e,1),:)';
    tangent4e(3,:,:) = tangents(2*size(n4e,1)+1:3*size(n4e,1),:)';
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.35: common/computeTangent4s.m

```
function tangent4s = computeTangent4s(c4n,n4s)
%% computeTangent4s - Tangents for sides.
```

```
% computeTangent4s(c4n, n4s) computes the tangent vector of each side of the
% decomposition. c4n and n4s are as specified in
% the documentation.
%
% See also: computeN4s, computeTangent4e, computeNormal4s

    if isempty(n4s)
        tangent4s = zeros(0,2);
        return;
    end

    %% Compute length4s.
    c4start = c4n(n4s(:,1),:);
    c4end = c4n(n4s(:,2),:);
    length4s = sqrt(sum((c4end-c4start).^2,2));

    %% Compute tangent4s.
    tangent4s = (c4end-c4start)./[length4s length4s];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.36: common/loadGeometry.m

```
function [c4n n4e n4sDb n4sNb] = loadGeometry(name, OPTRefinementLevel)
%% loadGeometry - load data for a mesh.
% [c4n n4e n4sDb n4sNb] = loadGeometry('name') loads the data structures for
% the mesh named 'name'. Optionally, the second parameter will
% cause the mesh to be refined a given number of times using the uniform
% red strategy.
% Example:
```

```
% [c4n n4e n4sDb n4sNb] = loadGeometry('LShape',2) loads the
% mesh called 'LShape' and refines it two times.

    %% Load the geometry data.
    c4n = load([name,'_c4n.dat']);
    n4e = load([name,'_n4e.dat']);
    n4sDb = load([name,'_n4sDb.dat']);
    n4sNb = load([name,'_n4sNb.dat']);

    %% Initial refinement.
    if nargin < 2
        OPTRefinementLevel = 0;
    end
    for i=1:OPTRefinementLevel
        [c4n,n4e,n4sDb,n4sNb] = refineUniformRed(c4n,n4e,n4sDb,n4sNb);
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.37: common/P0AveragingP1.m

```
function val = P0AveragingP1(c4n,n4e,sigma4e)
%% Averaging of a given P0 function to a P1 function
% c4n, n4e - a triangular mesh
% sigma4e - values of a pieceweise P0 function v on the mesh

    %% Initialisation
    d = size(sigma4e,2);
    area4n = computeArea4n(c4n,n4e);
    area4e = computeArea4e(c4n,n4e);
```

```
    %% Compute node values.
    weightedV = sigma4e.*(area4e*ones(1,d))*[eye(d),eye(d),eye(d)];
    I = n4e(:,[ones(1,d),2*ones(1,d),3*ones(1,d)]); % node indeces
    J = (ones(size(n4e,1),1)*(1:d))*[eye(d),eye(d),eye(d)]; % component indeces
    val = accumarray([I(:),J(:)],weightedV(:))./(area4n*ones(1,d));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.38: common/P0NormalJump.m

```
function jump4s = P0NormalJump(c4n,n4e,n4sDb,n4sNb,sigma4e,g)
%% P0NormalJump - computes jumps in normal direction
% c4n,n4e,n4sDb,n4sNb: mesh data
% sigma4e: the values of a P0 function given for each element
% g: a function giving values for the Neumann boundary
%
% jump4s: the jumps of v across each side, for Neumann boundary
% sides, g is the expected value.

    %% Initialisation
    s4e = computeS4e(n4e);
    s4n = computeS4n(n4e);
    normal4e = computeNormal4e(c4n,n4e);
    n4s = computeN4s(n4e);
    mid4s = computeMid4s(c4n, n4s);
    jump4s = zeros(size(n4s,1),1);

    %% inner jumps
```

```
    for elem = 1 : size(n4e,1)
        jump4s(s4e(elem,:)) = jump4s(s4e(elem,:)) ...
                            + normal4e(:,:,elem)*sigma4e(elem,:)';
    end

    %% Neumann jumps
    for nodes = n4sNb'
        side = s4n(nodes(1),nodes(2));
        jump4s(side) = jump4s(side) - g(mid4s(side,:));
    end

    %% Dirichlet jumps = 0
    jump4s(diag(s4n(n4sDb(:,1),n4sDb(:,2)))) = 0;

    jump4s = abs(jump4s);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.39: common/P0TangentJump.m

```
function jump4s = P0TangentJump(c4n,n4e,n4sDb,n4sNb,sigma4e,u4Db)
%% P0TangentJump - computes jumps in tangential direction
% c4n,n4e,n4sDb,n4sNb: mesh data
% sigma4e: the values of a P0 function given for each element
% u4Db: a function giving values for the Dirichlet boundary
%
% jump4s: the jumps of v across each side, for Dirichlet boundary
% sides, u4Db is the expected value.
```

```
    %% Initialisation
    s4e = computeS4e(n4e);
    s4n = computeS4n(n4e);
    n4s = computeN4s(n4e);
    tangent4e = computeTangent4e(c4n,n4e);
    length4s = computeLength4s(c4n,n4s);
    jump4s = zeros(size(n4s,1),1);

    %% inner jumps
    for elem = 1 : size(n4e,1)
        jump4s(s4e(elem,:)) = jump4s(s4e(elem,:)) ...
                            + tangent4e(:,:,elem)*sigma4e(elem,:)';
    end

    %% Neumann jumps
    jump4s(diag(s4n(n4sNb(:,1),n4sNb(:,2)))) = 0;

    %% Dirichlet jumps
    for nodes = n4sDb'
        side = s4n(nodes(1),nodes(2));
        jump4s(side) = jump4s(side) ...
            - (u4Db(c4n(nodes(2),:))-u4Db(c4n(nodes(1),:)))/length4s(side);
    end

    jump4s = abs(jump4s);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
```

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

Listing 3.40: common/estimateCREtaNormalOnly.m

```
function [eta4s,n4s] = estimateCREtaNormalOnly(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
%% estimateCREtaSides - error estimator for CR element
% Estimate the energy error of the CR finite element solution by the
% jumps of the discrete solution's tangents along the sides.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% x CR basis coefficients of u given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4s error for each side
% n4s nodes of the sides for which the error was computed

    %% Initialisation
    s4e = computeS4e(n4e);
    n4s = computeN4s(n4e);
    length4s = computeLength4s(c4n,n4s);
    gradU = zeros(size(n4e,1),2);

    %% Compute gradient.
    for elem = 1:size(n4e,1);
        grads = [c4n(n4e(elem,:),:)'; 1 1 1] \ [-2 0; 0 -2; 0 0];
        gradU(elem,:) = x(s4e(elem,:))' * grads([3 1 2],:);
    end

    %% Compute the L2-norm of the jumps and weigh them with length4s.
    eta4sTangent = P0TangentJump(c4n,n4e,n4sDb,n4sNb,gradU,u4Db);
    eta4s = (eta4sTangent.*length4s).^2;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
```

```
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.41: estimate/estimateCREtaSides.m

```
function [eta4s,n4s] = estimateCREtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
%% estimateCREtaSides - error estimator for CR element
% Estimate the energy error of the CR finite element solution by the
% jumps of the discrete solution's tangents along the sides.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% x CR basis coefficients of u given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4s error for each side
% n4s nodes of the sides for which the error was computed

    %% Initialisation
    s4e = computeS4e(n4e);
    n4s = computeN4s(n4e);
    length4s = computeLength4s(c4n,n4s);
    gradU = zeros(size(n4e,1),2);

    %% Compute gradient.
    for elem = 1:size(n4e,1);
        grads = [c4n(n4e(elem,:),:)'; 1 1 1] \ [-2 0; 0 -2; 0 0];
        gradU(elem,:) = x(s4e(elem,:))' * grads([3 1 2],:);
    end

    %% Compute the L2-norm of the jumps and weigh them with length4s.
    eta4sNormal = P0NormalJump(c4n,n4e,n4sDb,n4sNb,gradU,g);
    eta4sTangent = P0TangentJump(c4n,n4e,n4sDb,n4sNb,gradU,u4Db);
    eta4s = ((eta4sNormal+eta4sTangent).*length4s).^2;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
```

```
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.42: estimate/estimateCRP0Elements.m

```
function eta4e = estimateCRP0Elements(f,g,u4Db,u,p,c4n,n4e,n4sDb,n4sNb,mu)

    if nargin < 10
        mu = 1;
    end
    %% initialisation
    n4s = computeN4s(n4e);
    s4e = computeS4e(n4e);

    nrElems = size(n4e,1);

    area4e = computeArea4e(c4n,n4e);
    mid4e = computeMid4e(c4n,n4e);

    length4s = computeLength4s(c4n,n4s);

    %% Compute the CR-gradient of u
    grad4e1 = zeros(nrElems,2);
    grad4e2 = zeros(nrElems,2);
    for elem = 1 : nrElems
        grads = [1,1,1;c4n(n4e(elem,:),:)'] \ [0,0;eye(2)];
        nc_grads = [-1,1,1;1,-1,1;1,1,-1] * grads;
        grad4e1(elem,:) = u(s4e(elem,:),1)'*nc_grads;
        grad4e2(elem,:) = u(s4e(elem,:),2)'*nc_grads;
    end

    %% jumps
% jumpN1 = mu*P0NormalJump(c4n,n4e,n4sDb,n4sNb,grad4e1-[p,p],@(x)gNb(x,1,g));
% jumpN2 = mu*P0NormalJump(c4n,n4e,n4sDb,n4sNb,grad4e2-[p,p],@(x)gNb(x,2,g));
    jumpT1 = mu*P0TangentJump(c4n,n4e,n4sDb,n4sNb,grad4e1,@(x)uDb(x,1,u4Db));
    jumpT2 = mu*P0TangentJump(c4n,n4e,n4sDb,n4sNb,grad4e2,@(x)uDb(x,2,u4Db));
    eta4s = length4s .* (jumpT1.^2+jumpT2.^2);%+jumpN1.^2+jumpN2.^2);

    %% Volume term
```

```
    f = f(mid4e);
    eta4e = area4e .* (f(:,1).^2+f(:,2).^2);

    %% Output
    eta4e = sqrt(area4e.*eta4e + 1/2*sum(length4s(s4e).*eta4s(s4e),2));
end

function val = uDb(x,n,u4Db)
val = u4Db(x);
val = val(:,n);
end

function val = gNb(x,n,g)
val = g(x);
val = val(:,n);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.43: estimate/estimateL2Averaging.m

```
function [av4e] = estimateL2Averaging(c4n,n4e,p)
%% error4eAveragingL2 -
% Compute the averaging L2 error for a given finite element solution p of
% stress or flux.
%
% Input: c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% p discrete solution of stres or flux
%
```

```
% Output: av4e averaging L2 error on each element
%

    %% Initialization
    area4e=computeArea4e(c4n,n4e);
    mid4e=computeMid4e(c4n,n4e);
    p=sparse(p);


    %% Computation of averaging
    stack3=kron(ones(3,1),1:size(n4e,1));
    n4e2Area=sparse(stack3,n4e',[area4e; area4e; area4e]');
    Ap=(n4e2Area'*p(:,[1 2]))./(sum(n4e2Area,1)'*sparse([1 1]));

    % calculate nodal values
    tmp = reshape(p(stack3, [1 2]) - Ap(n4e',:) ...
    + p(stack3,[3 3]).*(c4n(n4e',:)-mid4e(stack3,:)),3,2*size(n4e,1));

    % calculate L2-integral
    av4e = sum(sum(repmat(area4e',3,2).*tmp.*tmp,1)...
    + repmat(area4e',1,2).*sum(tmp,1).*sum(tmp,1))/12;


end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.44: estimate/estimateNCStokesEtaElements.m

```
function [eta4e,n4s] = estimateNCStokesEtaElements(c4n,n4e,n4sDb,f,Du4Db1,Du4Db2,u,gradU4e)
```

```
%% estimateCREtaSides - error estimator for CR solution of Stokes problem
% Estimate the energy error of the CR finite element solution of the Stokes
% problem by the jumps of the discrete solution's tangents along the sides.
%
% Input: c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% f right-hand side of the problem definition
% Du4Db1 first row of the gradient of the Dirichlet boundary
% condition for computation of the normal derivative
% Du4Db2 second row of the gradient of the Dirichlet boundary
% condition for computation of the normal derivative
% u basis coefficients of the numerical solution of the
% velocity field w.r.t. the Crouzeix-Raviart basis
% gradU4e piecewise gradient of the discrete solution u
%
% Output: eta4e error for each element
% n4s vector which contains in one row for each side the
% numbers of the corresponding nodes

    %% Initialization
    s4e=computeS4e(n4e);
    n4s=computeN4s(n4e);
    s4n=computeS4n(n4e);
    tangent4e=computeTangent4e(c4n,n4e);
    tangent4s=computeTangent4s(c4n,n4s);
    nrElems=size(n4e,1);
    nrSides=size(n4s,1);
    length4s=computeLength4s(c4n,n4s);
    area4e=computeArea4e(c4n,n4e);

    %% Compute gradient.
    if nargin < 8
        gradU4e=zeros(2,2,nrElems);
        for j=1:nrElems
            grads=[c4n(n4e(j,:),:)'; 1 1 1]\[-2 0; 0 -2; 0 0];
            gradU4e(1,:,j)=u(s4e(j,:),1)' * grads([3 1 2],:);
            gradU4e(2,:,j)=u(s4e(j,:),2)' * grads([3 1 2],:);
        end
    end

    %% Compute inner jumps
    jump4s=zeros(nrSides,2);
    for j=1:nrElems
           jump4s(s4e(j,:),:) = jump4s(s4e(j,:),:) ...
                                + [tangent4e(:,:,j)*gradU4e(1,:,j)' ...
                                   tangent4e(:,:,j)*gradU4e(2,:,j)'];
    end

    %% Dirichlet jumps
    l4DbS=computeLength4s(c4n,n4sDb);
    mean4DbSides1=integrate(c4n,n4sDb,@(x,y,z)(Du4Db1(y)),10)...
```

```
                                                    ./[l4DbS l4DbS];
    mean4DbSides2=integrate(c4n,n4sDb,@(x,y,z)(Du4Db2(y)),10)...
                                                    ./[l4DbS l4DbS];
    for j=1:size(n4sDb,1)
        nodes=n4sDb(j,:)';
        side=s4n(nodes(1),nodes(2));
        jump4s(side,:) = jump4s(side,:) ...
                         - [mean4DbSides1(j,:)*tangent4s(side,:)'...
                            mean4DbSides2(j,:)*tangent4s(side,:)'];
    end

    %% Compute eta4e
    jump4s_L2normSq=(jump4s(:,1).^2 + jump4s(:,2).^2) .* length4s;
    [~,L2normSq4e]=L2Norm(c4n,n4e,@(x,y,z)f(y),6);
    eta4e = area4e.*sum(L2normSq4e,2) +...
                            sqrt(area4e).*sum(jump4s_L2normSq(s4e),2);

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
```

```
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.45: estimate/estimateP1AveragingP1.m

```
function eta4e = estimateP1AveragingP1(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
%% estimateP1AveragingP1 - averaging error estimator for P1 element
% Estimate the gradient error of the P1 finite element solution by
% comparing the P0-gradient of the discrete solution to the P1 average
% of itself.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% x P1 basis coefficients of u given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4e error for each element

    %% Initialisation
    nrElems = size(n4e,1);
    grad4e = zeros(nrElems,2);

    %% Compute the P0-gradient of u
    for elem = 1 : nrElems
        grads = [1,1,1;c4n(n4e(elem,:),:)'] \ [0,0;eye(2)];
        grad4e(elem,:) = x(n4e(elem,:))'*grads;
    end

    eta4e = estimateSigmaAveragingP1(c4n,n4e,grad4e);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
```

```
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.46: estimate/estimateP1EtaElements.m

```
function eta4e = estimateP1EtaElements(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
% estimateP1EtaSides.m
% Estimate the energy error of the P1 finite element solution by the
% jumps of the discrete solution over the sides and a volume term.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% u solution u given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4e error for each element

    %% initialisation
    s4e = computeS4e(n4e);
    area4e = computeArea4e(c4n,n4e);
    mid4e = computeMid4e(c4n,n4e);

    %% Compute eta4s.
    eta4s = estimateP1EtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb);

    %% Compute eta4e.
    eta4e = (area4e.*f(mid4e)).^2 + sum(eta4s(s4e),2);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
```

```
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.47: estimate/estimateP1EtaSides.m

```
function [eta4s,n4s] = estimateP1EtaSides(f,g,u4Db,x,c4n,n4e,n4sDb,n4sNb)
%% estimateP1EtaSides.m
% Estimate the energy error of the P1 finite element solution by the
% jumps of the discrete solution over the sides.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% u solution u given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4s error for each side
% n4s nodes of the sides for which the error was computed

    %% Initialisation
    n4s = computeN4s(n4e);
    length4s = computeLength4s(c4n,n4s);
    nrElems = size(n4e,1);
    gradU = zeros(nrElems,2);

    %% Compute grad U
    for elem = 1 : nrElems
        grads = [1,1,1;c4n(n4e(elem,:),:)']\[0,0;eye(2)];
        gradU(elem,:) = x(n4e(elem,:))' * grads;
    end
```

```
    %% Compute the L2-norm of the jumps and weigh them with length4s
    eta4s = (P0NormalJump(c4n,n4e,n4sDb,n4sNb,gradU,g).*length4s).^2;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.48: estimate/estimateRT0EtaSides.m

```
function [eta4s,n4s] = estimateRT0EtaSides(f, g, u4Db, p, u, c4n, n4e, n4sDb, n4sNb)
%% estimateRT0EtaSides - error estimator for RT0 element
% Estimate the energy error of the RT0 finite element solution by the
% jumps of the discrete solution's P0 component along the sides.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% x RT0 basis coefficients of (u,p) given by solve
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: eta4s error for each side
% n4s nodes of the sides for which the error was computed

    %% initialisation
    s4n = computeS4n(n4e);
    n4s = computeN4s(n4e);
    e4s = computeE4s(n4e);

    length4s = computeLength4s(c4n,n4s);
```

```
    area4e = computeArea4e(c4n,n4e);

    if size(n4sDb,1)>0
        s4Db = diag(s4n(n4sDb(:,1),n4sDb(:,2)));
    else
        s4Db = [];
    end

    s4Nb = diag(s4n(n4sNb(:,1),n4sNb(:,2)));

    %% jump and volume term
    jump4s = integrate(c4n, n4s, @(parts, Gpts4p, Gpt4ref)...
         computeJump4s(parts, Gpts4p, Gpt4ref, p, n4e, c4n,...
         e4s, s4Db, s4Nb, n4s, length4s, u4Db), 2);

    vol4e = integrate(c4n, n4e, @(parts, Gpts4p, Gpt4ref) f(Gpts4p).^2, 2);

    TPlus4s = e4s(:,1);
    TMinus4s = e4s(:,2);
    vol4eTMinus = zeros(size(n4s,1),1);
    vol4eTMinus(TMinus4s(TMinus4s~=0)) = vol4e(TMinus4s(TMinus4s~=0));
    area4eTMinus = zeros(size(n4s,1),1);
    area4eTMinus(TMinus4s(TMinus4s~=0)) = area4e(TMinus4s(TMinus4s~=0));

    eta4s = length4s .* jump4s + area4e(TPlus4s).*vol4e(TPlus4s)/3 ...
           + area4eTMinus.*vol4eTMinus/3;

end

%% Jumps
function jumps4s = computeJump4s(n4parts, Gpts4p, Gpt4ref, p, n4e, c4n, ...
                             e4s,s4Db,s4Nb, n4s, length4s, u4Db)
    tangent4s = computeTangent4s(c4n,n4parts);
    mid4e = computeMid4e(c4n, n4e);

    % components of the solution
    p1 = p(:,1);
    p2 = p(:,2);
    p3 = p(:,3);

    TPlus4s = e4s(:,1);
    TMinus4s = e4s(:,2);
    %Use element 1 for calculation and overwrite that later with the correct value
    TMinus4s(TMinus4s == 0) = 1;

    valTPlus = sum ( (p1(TPlus4s) * [1 0] + p2(TPlus4s) * [0 1] ...
                    + (Gpts4p - mid4e(TPlus4s,:)) ...
                    .* [p3(TPlus4s) p3(TPlus4s)]).* tangent4s, 2);
    valTMinus = sum ( (p1(TMinus4s) * [1 0] + p2(TMinus4s) * [0 1] ...
                    + (Gpts4p - mid4e(TMinus4s,:)) ...
                    .* [p3(TMinus4s) p3(TMinus4s)]).* tangent4s, 2);
    % p' in t
```

```
    valTMinus(s4Db) = (u4Db(c4n(n4s(s4Db,2),:)) - u4Db(c4n(n4s(s4Db,1),:))) ...
                  ./ length4s(s4Db);
    jumps4s = (valTPlus - valTMinus) .^ 2;
    jumps4s(s4Nb) = 0;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.49: estimate/estimateSigmaAveragingP1.m

```
function eta4e = estimateSigmaAveragingP1(c4n,n4e,sigma4e)
%% estimateSigmaAveragingP1 - L2 difference of sigma and A(sigma)
% Given an elementwise constant function sigma on a triangulation
% ompute the L2 norm of A(sigma)-sigma.
% Input: c4n,n4e: triangulation
% sigma4e: values of sigma for each element
%
% Output: eta4e: ||A(sigma)-sigma||_L2(T) for each element T

    %% Initialisation
    nrElems = size(n4e,1);
    area4e = computeArea4e(c4n,n4e);
    eta4e = zeros(nrElems,1);

    %% Compute the P1 average of sigma
    A = P0AveragingP1(c4n,n4e,sigma4e);

    %% Eta for each element
    for elem = 1 : nrElems
        s=0.5*[1 1 0;0 1 1;1 0 1]*A(n4e(elem,:),:)-[1;1;1]*sigma4e(elem,:);
```

```
        eta4e(elem) = sum(sum(s.^2))*area4e(elem)/3;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.50: integrate/error4eCREnergy.m

```
function val = error4eCREnergy(c4n,n4e,gradExact,uApprox)
    nrElems = size(n4e,1);
    s4e = computeS4e(n4e);
    gradUApprox = zeros(nrElems,2);

    % Compute gradient for x
    for elem = 1:size(n4e,1);
        grads = [c4n(n4e(elem,:),:)'; 1 1 1] \ [-2 0; 0 -2; 0 0];
        gradUApprox(elem,:) = uApprox(s4e(elem,:))' * grads([3 1 2],:);
    end

    val = sum(integrate(c4n,n4e, ...
        @(n4p, Gpts4p, Gpts4ref) (gradExact(Gpts4p) - gradUApprox).^2, 6),2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
```

```
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.51: integrate/error4eCRL2.m

```
function error4e = error4eCRL2(c4n,n4e,uExact,uApprox)

s4p = computeS4e(n4e);

error4e = (integrate(c4n,n4e, ...
    @(n4p, Gpts4p, Gpts4ref) ((uExact(Gpts4p)-...
        ((-1 + 2*Gpts4ref(1) + 2*Gpts4ref(2))*uApprox(s4p(:,2)) +...
        (1-2*Gpts4ref(1))*uApprox(s4p(:,3)) +...
        (1-2*Gpts4ref(2))*uApprox(s4p(:,1)))).^2), 12));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.52: integrate/error4eP1Energy.m

```
function error4e = error4eP1Energy(c4n,n4e,gradExact,uApprox)
%% error4eP1Energy - compute the energy error on elements
% Input: c4n,n4e - mesh
% gradExact - the exact gradient
% uApprox - AFEM P1 solution
%
% Output: error4e - the exact squared energy error on each element


    %% Compute grad U
    nrElems = size(n4e,1);
    gradU = zeros(nrElems,2);

    for elem = 1 : nrElems
        grads = [1,1,1;c4n(n4e(elem,:),:)']\[0,0;eye(2)];
        gradU(elem,:) = uApprox(n4e(elem,:))' * grads;
    end

    %% Compute the error

    error4e = integrate(c4n,n4e,@(n4p, Gpts4p, Gpts4ref) (...
              sum((gradExact(Gpts4p) - gradU ).^2,2)),6);


end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.53: integrate/error4eP1L2.m

```
function error4e = error4eP1L2(c4n,n4e,uExact,uApprox)
%% error4eP1L2 - compute the exact error on elements
% Input: c4n,n4e - mesh
% uExact - the exact function
% uApprox - AFEM P1 solution
%
% Output: error4e - the exact squared error on each element

%% Compute error
  error4e = (integrate(c4n,n4e,@(n4p, Gpts4p, Gpts4ref) (...
                uExact(Gpts4p)-...
             ((1 - Gpts4ref(:,1) - Gpts4ref(:,2))*uApprox(n4p(:,1)) +...
             (Gpts4ref(:,1))*uApprox(n4p(:,2)) +...
             (Gpts4ref(:,2))*uApprox(n4p(:,3)))).^2,4));

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.54: integrate/error4eRT0Energy.m

```
function val = error4eRT0Energy(c4n, n4e, gradExact, pApprox)
%% error4eRT0energy - energy error for RT0 element
% Calculates the energy error of the RT0 finite element solution with given
% exact gradiant function gradExact.
%
% Input: c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% p first component of the solution
% u second component of the solution
```

```
% gradExact exact gradiant of the solution
%
% Output: val L2 error of the gradiant on each element.

val = sum(...
    integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)...
    (pApprox(:,1)*[1 0] + pApprox(:,2)*[0 1]...
    + [pApprox(:,3) pApprox(:,3)] .* (Gpts4p - computeMid4e(c4n,n4p)) ...
    - gradExact(Gpts4p)).^2, 6)...
    ,2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.55: integrate/error4eRT0L2.m

```
function val = error4eRT0L2(c4n, n4e, uExact, uApprox)
%% error4eRT0energy - energy error for RT0 element
% Calculates the L2 error of the RT0 finite element solution with given
% exact solution uExact.
%
% Input: c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% p first component of the solution
% u second component of the solution
% uExact exact gradiant of the solution
%
% Output: val L2 error of the solution u on each element.

val = integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)...
    (uApprox - uExact(Gpts4p)).^2,2);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.56: integrate/error4eStokesCRStress.m

```
function error4e = error4eStokesCRStress(c4n,n4e,component,stressExact,uApprox,pApprox,gradUApp
%% error4eStokesCRStress - exact stress error of CR Stokes solution
% Compute the exact error of one row of the stress tensor for a given CR
% finite element solution of the Stokes problem.
%
% Input: c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% component number of row of the stress tensor for which the
% error should be computed
% stressExact exact stress function
% uApprox basis coefficients of the numerical solution of the
% velocity field w.r.t. the Crouzeix-Raviart basis
% pApprox basis coefficients of the numerical solution of the
% pressure w.r.t. the P0 basis
% gradUApprox corresponding row of the piecewise gradient of the
% discrete solution u (OPTIONAL)
%
% Output: error4e exact stress error on each element

    %% Initialization
    nrElems = size(n4e,1);
    s4e = computeS4e(n4e);

    %% Computation of discrete gradient (optional)
    if nargin < 7
        gradUApprox = zeros(nrElems,2);
```

```
        for j = 1:nrElems;
            grads = [c4n(n4e(j,:),:)'; 1 1 1] \ [-2 0; 0 -2; 0 0];
            gradUApprox(j,:) = uApprox(s4e(j,:))' * grads([3 1 2],:);
        end
    end

    %% Computation of discrete stress
    if component == 1
        stressUApprox = gradUApprox-[pApprox,zeros(size(gradUApprox,1),1)];
    elseif component == 2
        stressUApprox = gradUApprox-[zeros(size(gradUApprox,1),1),pApprox];
    end

    %% Computation of error
    error4e = sum(integrate(c4n,n4e,@(n4p,Gpts4p,Gpts4ref)...
                          (stressExact(Gpts4p)-stressUApprox).^2, 6),2);

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
```

```
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.57: integrate/integrate.m

```
function val = integrate(c4n, n4p, integrand, degree, OPTsize4parts)
% Integrates function handle integrand over given domain 1D or 2D.
% 'integrand' must have the form integrand(n4p,Gpts4p,Gpts4ref) and the
% return value must have the dimension [ nrParts n m ]
%
% integrate.m
% input: c4n - coordinates for nodes
% n4p - nodes of the parts of a partition of a 1D or 2D domain
% integrand - function to integrate
% degree - integration is exact for polynomials up to degree
% OPTsize4parts - length4s (1D) or area4e (2D) (Optional parameter)
% output: val - integral value of dimension [ nrParts n m ]

% [val of dimension (nrParts x n x m)] = integrand(n4p,Gpts4p,Gpts4ref)
% input: n4p - nodes of the parts of a partition of a 1D or 2D domain
% Gpts4p - coordinates of the transformed GaussPoints on each part
% Gpts4ref - coordinates of the GaussPoints of the reference intervall
% or the reference triangle

    %% Number of Parts
    nrParts = size(n4p,1);

    %% Number of Gauss Points
    nrGpts = ceil((degree+1)/2);

    %% 1D or 2D
    if size(n4p,2) == 2
        % 1D
        [Gpts4ref,Gwts4ref] = getGaussPoints(nrGpts);
        if nargin == 5
            weightFactor = OPTsize4parts;
        else
            weightFactor = computeLength4s(c4n,n4p);
        end
```

```
    elseif size(n4p,2) == 3
        % 2D triangle
        [Gpts4ref,Gwts4ref] = getConProdGaussPoints(nrGpts);
        if nargin == 5
            weightFactor = 2*OPTsize4parts;
        else
            weightFactor = 2*computeArea4e(c4n,n4p);
        end
        nrGpts = nrGpts^2;

    else
        error('Could not integrate because no Domain is specified.');
    end
    % Integrate
    for curGpt = 1:nrGpts
        %transformation
        curGpt4p = ref2arbitrary(c4n,n4p,Gpts4ref(curGpt,:));
        % evaluate function handle eval must be [nrParts n m]
        integrandVal = integrand(n4p,curGpt4p,Gpts4ref(curGpt,:));
        if(curGpt == 1)
            dim = size(integrandVal);
            if numel(dim) < 3; dim(3) = 1; end
            val = zeros(nrParts,dim(2),dim(3));
        end
        % weighted sum
        curGwt4p = weightFactor*Gwts4ref(curGpt);
        curGwt4p = reshape( curGwt4p*ones(1,dim(2)*dim(3)),[nrParts,dim(2),dim(3)]);
        val = val + curGwt4p.*integrandVal;
    end
end


%% ref2arbitrary %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function val = ref2arbitrary(c4n,n4p,curGpt)
    %1D or 2D
    if size(n4p,2) == 2
        %1D ref = conv(0,1)
        c1 = c4n(n4p(:,1),:);
        c2 = c4n(n4p(:,2),:);

        val = c1 + curGpt(1)*(c2-c1);

    elseif size(n4p,2) == 3
        % 2D ref = conv{ (0,0),(1,0),(0,1)}
        c1 = c4n(n4p(:,1),:);
        c2 = c4n(n4p(:,2),:);
        c3 = c4n(n4p(:,3),:);

        val = c1 + curGpt(1)*(c2-c1) + ...
                  curGpt(2)*(c3-c1);
    end
```

```
end

function [x,w] = getGaussPoints(n)
% Gives n Gauss points for the unite vector.
% Used for Gauss-Legendere integration.
%
% input: n - number of points
% output: x - Gauss points
% w - Gauss weights

    % Find n Gauss_Legendre Points for Intervall [-1,1]
    gamma = (1 : n-1) ./ sqrt(4*(1 : n-1).^2 - ones(1,n-1) );
    [V,D] = eig( diag(gamma,1) + diag(gamma,-1) );
    x = diag(D);
    w = 2*V(1,:).^2;

    % linear map to Intervall [0,1]
    x = .5 * x + .5;
    w = .5 * w';
end

function [x,w] = getConProdGaussPoints(n)
% Gauss Points for the Reference Triangle
% T = {x+y| 0<=x<=1, 0<=y<=1, x+y<=1}
% Integrates polynomials up to degree 2n-1 exact
% using the Stroud Conical Product rule with n^2
% quadrature Points.
%
% input: n - number of points
% output: x - Gauss points
% w - Gauss weights

    % Find n Gauss_Legendre Points for Intervall [-1,1]
    gamma = (1 : n-1) ./ sqrt(4*(1 : n-1).^2 - ones(1,n-1) );
    [V,D] = eig( diag(gamma,1) + diag(gamma,-1) );
    r = diag(D);
    a = 2*V(1,:).^2;% norm-factor -> int(1,-1,1)=2

    % Find n Gauss_Jacobi Points for Intervall [-1,1]
    delta = -1./(4*(1 : n).^2-ones(1,n));
    gamma = sqrt((2 : n).*(1 : n-1)) ./ (2*(2 : n)-ones(1,n-1));
    [V,D] = eig( diag(delta)+diag(gamma,1)+diag(gamma,-1) );
    s = diag(D);
    b = 2*V(1,:).^2; % norm-factor -> int((1-x),-1,1)=2

    % linear map to Intervall [0,1]
    % w(x) = 1 changes norm-factor from 2 to 1
    r = .5 * r + .5;
    a = .5 * a';
    % w(x) = (1-x) changes norm-factor from 2 to 1/2
    s = .5 * s + .5;
    b = .25 * b';
```

```
    % conical product [ s_j , r_i(1-s_j) ] a_i*b_j
    s = repmat(s',n,1); s = s(:);
    r = repmat(r,n,1);
    x = [ s , r.*(ones(n^2,1)-s) ];
    w = a*b';
    w = w(:);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.58: integrate/L2Norm.m

```
function [L2norm4Omega, L2norm4p] = L2Norm(c4n,n4e,f,degree)
%% L2Norm - compute the L2 norm of a function on a triangulation
% Input: c4n,n4e - mesh
% f - function, the norm of which is computed;
% has to suit input/output behaviour needed by integrate
% degree - degree of accuracy used in the quadrature formula
% Output: L2norm4p - ||f||_(L^2(Part))
% L2normOmega - ||f||_(L^2(Omega))

    %% Compute the L2 norm.
    L2norm4p = integrate(c4n, n4e, ...
               @(n4p,Gpts4p,Gpts4ref)(f(n4p,Gpts4p,Gpts4ref)).^2, degree);
    L2norm4Omega = sqrt(sum(L2norm4p));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
```

```
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.59: integrate/oscillations.m

```
function [osc4e ,mean4e] = oscillations(c4n,n4e,f,degree)
%% oscillcations - oscillations of a function on a mesh
% Input: c4n,n4e - mesh
% f - R^2->R; input: points; output: values
% degree - accuracy of integration
% Output: osc4e - vector of oscillations squared for each element
% mean4e - vector of integral means of f for each element
%% Initialisation
    meanDeg = degree;
    oscDeg = 2*degree;
    area4e = computeArea4e(c4n,n4e);
%% Compute the integral mean of f on each element.
    mean4e = integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)(f(Gpts4p)),meanDeg)...
            ./area4e;
%% Compute oscillations: locally on each T - osc4e = ||f-mean(f)||_L2(T) / |T|
    osc4e = integrate(c4n, n4e, @(n4p,Gpts4p,Gpts4ref)((f(Gpts4p)-mean4e).^2),...
            oscDeg);
    osc4e = osc4e .* area4e;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
```

```
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.60: mark/markBulk.m

```
function n4sMarked = markBulk(n4p,eta4p,OPTtheta)
%% markBulk - Mark given parts using the bulk criterion.
% n4sMarked = markBulk(n4p, eta4p, OPTtheta) marks the parts with the
% largest estimated errors. The aggregate error of the marked parts
% is OPTtheta times the overall error. By default, OPTtheta is set
% to 0.5. n4p contains the nodes for the parts: 2 entries per row
% for sides, 3 entries per row for elements.
% The output is a list of marked sides given by their end nodes.

    if (nargin < 3)
        theta = 0.5;
    else
        theta = OPTtheta;
    end
    dimParts = size(n4p,2);

    %% Bulk criterion
    [eta4p,ind] = sort(eta4p,'descend');
    % avoid round-off errors between sum and cumsum (esp. if theta=1)
    cumsumEta4p = cumsum(eta4p);
    J = find(cumsumEta4p >= theta*cumsumEta4p(end),1,'first');
    I = ind(1:J);

    %% Mark sides
    if dimParts == 2 % Sides were given. Mark 'em all.
        n4sMarked = n4p(I,:);
    elseif dimParts == 3 % Elements were given. Mark all sides of these.
        allSidesMarked = [n4p(I,[1 2]);n4p(I,[2 3]);n4p(I,[3 1])];
        % Eliminate duplicates.
        [b, ind] = unique(sort(allSidesMarked,2), 'rows');
        n4sMarked = allSidesMarked(sort(ind),:);
    end
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.61: mark/markMaximum.m

```
function n4sMarked = markMaximum(n4p, eta4p, OPTtheta)
%% markMaximum - Mark given parts using the maximum criterion.
% n4sMarked = markMaximum(n4p, eta4p, OPTtheta) marks all given parts for
% which eta4p >= OPTtheta*max(eta4p) is satisfied. n4p contains the
% nodes for the parts: 2 entries per row for sides, 3 entries per row
% for elements. By default, OPTtheta is set to 0.5.
% The output is a list of marked sides given by their end nodes.

    if (nargin < 3)
        theta = 0.5;
    else
        theta = OPTtheta;
    end
    dimParts = size(n4p,2);

    %% Maximum criterion
    I = find(eta4p >= theta*(max(eta4p)));

    %% Mark sides
    if dimParts == 2 % Sides were given. Mark 'em all.
        n4sMarked = n4p(I,:);
    elseif dimParts == 3 % Elements were given. Mark all sides of these.
        allSidesMarked = [n4p(I,[1 2]);n4p(I,[2 3]);n4p(I,[3 1])];
        % Eliminate duplicates.
```

```
        [b, ind] = unique(sort(allSidesMarked,2), 'rows');
        n4sMarked = allSidesMarked(sort(ind),:);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.62: mark/markUniform.m

```
function n4sMarked = markUniform(n4p)
%% markUniform - Mark given parts uniformly.
% n4sMarked = markUniform(n4p) marks every given part for refinement,
% regardless of estimated errors. n4p contains the nodes for the
% parts: 2 entries per row for sides, 3 entries per row for elements.
% The output is a list of marked sides given by their end nodes.

    nrParts = size(n4p,1);
    dimParts = size(n4p,2);

    %% Uniform criterion: mark everything
    I = 1:nrParts;

    %% Mark sides
    if dimParts == 2 % Sides were given. Mark 'em all.
        n4sMarked = n4p(I,:);
    elseif dimParts == 3 % Elements were given. Mark all sides of these.
        allSidesMarked = [n4p(I,[1 2]);n4p(I,[2 3]);n4p(I,[3 1])];
        % Eliminate duplicates.
        [b, ind] = unique(sort(allSidesMarked,2), 'rows');
        n4sMarked = allSidesMarked(sort(ind),:);
```

```
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.63: plot/plotConvergence.m

```
function plotConvergence(nrDoF4lvl, error4lvl, OPTname)
%% Plot error for different levels.
% plotConvergence(nrDoF4lvl, error4lvl) plots the error given by
% error4lvl over the degrees of freedom given by nrDoF4lvl, both
% using logarithmic scale. The optional String argument together
% with the convergence rate is added to the legend.
    if nargin > 2
        name = OPTname;
    else
        name = '';
    end

    cOrder = [0 0 1; 1 0 0; 0 .7 0; .7 0 .7; .7 .7 0; 0 0 0];

    nlines = get(gcf, 'UserData');

    if ~isempty(nlines)
      nlines = nlines + 1;
    else
      nlines = 1;
    end
    set(gcf, 'UserData', nlines);

    %% Plot.
```

```
    plot = loglog(nrDoF4lvl, error4lvl, '-s', 'Color', cOrder(nlines, :));
    title('Convergence history plot');

    %% Set name and title of the figure.
    set(plot,'DisplayName',name);

    legend('off');
    legend('show');
    handle = findobj(gcf,'type','axes','Tag','legend');
    set(handle,'Location','best');

    drawnow;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.64: plot/plotCR.m

```
function plotCR(c4n, n4e, x, OPTtitle)
%% Draw a Crouzeix-Raviart-function.
% plotCR(c4n, n4e, x, OPTtitle) draws the CR-function defined by the grid
% (c4n, n4e) and the basis coefficients (x)
% The input argument OPTtitle is optional,
% it sets the title of the figure. The
% default value is empty.

    if nargin == 4
            title(OPTtitle);
        else
```

```
12            title('');
    end

    %% Get coordinates for nodes.
    X1 = c4n(n4e(:,3),1);
17  Y1 = c4n(n4e(:,3),2);
    X2 = c4n(n4e(:,1),1);
    Y2 = c4n(n4e(:,1),2);
    X3 = c4n(n4e(:,2),1);
    Y3 = c4n(n4e(:,2),2);
22
    %% Translate values for degrees of freedom into values for nodes.
    s4e = computeS4e(n4e);
    W = x(s4e)'; % Get x for each side of each element.
    Z = (ones(3)-2*eye(3))*W;
27
    %% Assemble parameters for the patch function.
    X = [X1'; X2'; X3'];
    Y = [Y1'; Y2'; Y3'];
    % The colour of a triangle is determined by its midpoint.
32  C = sum(Z,1)/3;

    %% Put everything together.
    % For large numbers of elements, omit the black boundary around each
    % triangle.
37  if( size(n4e,1) > 2000 )
        patch(X,Y,Z,C,'EdgeColor','none');
    else
        patch(X,Y,Z,C);
    end
42  view(-37.5,30);
    grid on;
    drawnow;
end


47
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
52 % Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
57 % This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
62 % (at your option) any later version.
%
```

```
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.65: plot/plotCRP0.m

```
function plotCRP0(c4n, n4e, u, p)
%% plotCRP0 - plot a discrete solution for CR-P0 finite elements
% Input: c4n, n4e - mesh
% u - basis coefficients for (CR)^2 basis functions
% p - pressure on each element (optional)

    %% Plot p.
    if nargin > 3
        hold on;
        CData = zeros(max(n4e(:)), 3);
        CData(n4e(:, 1), :) = 0.5 + 0.5*(max(p)-p(:))/(max(p)-min(p))*[1 1 1];
        patch('Vertices', c4n, 'Faces', n4e, ...
              'FaceVertexCData', CData, 'FaceColor', 'flat');
    end

    %% Plot u.
    mid4s = computeMid4s(c4n,computeN4s(n4e));
    quiver2(mid4s(:, 1), mid4s(:, 2), u(:, 1), u(:, 2), ...
           'n=', 0.1, 'w=', [1 1]);
    axis equal tight;
    title({'Discrete flux'; [num2str(numel(u) + size(n4e, 1)), ' dofs']});
    drawnow;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
```

```
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.66: plot/plotEtaSidesTeach.m

```
function plotEtaSidesTeach(figure, c4n, n4s, eta4s)
%% Draw the estimated Error on sides.
% plotEtaSides(figure, c4n, n4s, eta4s) draws the estimated Error defined
% by the grid (c4n, n4s) and the error for each side
% into the given figure.

    angle = [-28, 56]; % set the point of view

    clf;
    [errX,errY,errZ,errC] = getErrorXYZC(c4n, n4s, eta4s);
    patch(errX,errY,errZ,errC);

    set(figure,'Name','AFEM-Error','Position',[670 500 600 400]);
    title('Estimated error on sides');
    view(angle);
    drawnow;
end

%% Function to get the coordinates for the patch-function
function [valX, valY, valZ,valC] = getErrorXYZC(c4n, n4s, eta4s)
    % coordinates for all nodes of sides
    X1 = c4n(n4s(:,1), 1);
    X2 = c4n(n4s(:,2), 1);
    Y1 = c4n(n4s(:,1), 2);
    Y2 = c4n(n4s(:,2), 2);

    % sides for the error in patch-style
    valX = [X1';X1';X2';X2'];
    valY = [Y1';Y1';Y2';Y2'];

    % each side should have the same height - the error
    valZ = [zeros(size(eta4s'));eta4s';eta4s';zeros(size(eta4s'))];
    valC = valZ / max(eta4s);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
```

```
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.67: plot/plotP04e.m

```
function plotP04e(c4n, n4e, x, OPTtitle)
%% Draw the P0-function on elements.
% plotP04e(c4n, n4e, x, OPTtitle) draws the P0-function defined by
% the grid (c4n, n4e) and the coefficient
% vector x into the given figure. The
% input argument OPTtitle is optional, it
% sets the title of the figure. The
% default value is empty.

    angle = [-37.5, 30];

    [errX,errY,errZ,errC] = getErrorXYZC(c4n, n4e, x);
    if(size(n4e,1)>2000)
        patch(errX,errY,errZ,errC,'EdgeColor','none');
    else
        patch(errX,errY,errZ,errC);
    end


    if nargin == 4
            title(OPTtitle);
        else
            title('');
    end

    view(angle);
    grid on;
    drawnow;
end

%% Function to get the coordinates for the patch-function
function [valX, valY, valZ,valC] = getErrorXYZC(c4n, n4e, x)
    % coordinates for all nodes of sides
    X1 = c4n(n4e(:,1), 1);
```

```
    X2 = c4n(n4e(:,2), 1);
    X3 = c4n(n4e(:,3), 1);
    Y1 = c4n(n4e(:,1), 2);
    Y2 = c4n(n4e(:,2), 2);
    Y3 = c4n(n4e(:,3), 2);

    % triangles for the function in patch-style
    valX = [X1';X2';X3';X3'];
    valY = [Y1';Y2';Y3';Y3'];

    % each element should have constant height - the function value
    valZ = [x';x';x';x'];
    valC = valZ / max(x);

    % create walls.
    nrElems = size(n4e,1);
    n4s= [n4e(:,1:2);n4e(:,2:3); n4e(:,3),n4e(:,1)];
    X = reshape(c4n(n4s',1),2,3*nrElems);
    Y = reshape(c4n(n4s',2),2,3*nrElems);
    X = [X; X(2,:); X(1,:)];
    Y = [Y; Y(2,:); Y(1,:)];
    Z = [1; 1; 0; 0]*[x', x', x'];

    valX = [valX, X];
    valY = [valY, Y];
    valZ = [valZ, Z];
    valC = [valC, Z/max(x)];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
```

```
87 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.68: plot/plotP04s.m

```
1  function plotP04s(c4n, n4e, x, OPTtitle)
   %% Draw the P0-function on sides.
3  % plotP04s(c4n, n4s, x, OPTtitle) draws the P0-function defined
   % by the grid (c4n, n4s) and the coeffi-
   % cient vector x into the active figure and
   % The input argument OPTtitle is optional, it
   % sets the title of the figure. The
8  % default value is empty.

       angle = [-37.5, 30]; % set the point of view


13     n4s = computeN4s(n4e);
       [errX,errY,errZ,errC] = getErrorXYZC(c4n, n4s, x);
       if(size(n4e,1)>2000)
           patch(errX,errY,errZ,errC,'EdgeColor','none');
       else
18         patch(errX,errY,errZ,errC);
       end

       if nargin == 4
              title(OPTtitle);
23         else
              title('');
       end

       view(angle);
28     grid on;
       drawnow;
   end

   %% Function to get the coordinates for the patch-function
33 function [valX, valY, valZ,valC] = getErrorXYZC(c4n, n4s, x)
       % coordinates for all nodes of sides
       X1 = c4n(n4s(:,1), 1);
       X2 = c4n(n4s(:,2), 1);
       Y1 = c4n(n4s(:,1), 2);
38     Y2 = c4n(n4s(:,2), 2);

       % sides for the function in patch-style
       valX = [X1';X1';X2';X2'];
       valY = [Y1';Y1';Y2';Y2'];
43
       % each side should have the same height - the function value
       valZ = [zeros(size(x'));x';x';zeros(size(x'))];
       valC = valZ / max(x);
   end
48
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.69: plot/plotP1.m

```
function plotP1(c4n, n4e, x, OPTtitle)
%% Draw a P1-function.
% plotP1(c4n, n4e, x, OPTtitle) draws the P1-function defined by the grid
% (c4n, n4e) and the basis coefficients (x).
% The input argument OPTtitle is
% optional, it sets the title of the figure.
% The default value is empty.

    if(size(n4e,1)>2000)
        trisurf(n4e,c4n(:,1),c4n(:,2),x,'EdgeColor','none');
    else
        trisurf(n4e,c4n(:,1),c4n(:,2),x);
    end

    if nargin == 4
            title(OPTtitle);
        else
            title('');
    end
    drawnow;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
```

```
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.70: plot/plotP1P1.m

```
function plotP1P1(n4e,c4n,x,lambda,mu,factor)
%% SHOW Plots two-dimensional solution
% SHOW(n4e,c4n,x,lambda,mu,factor) plots the
% strained mesh and visualizes the stresses in grey tones.
%
% The variable AVS is previously determined by the function <avmatrix.m>.
%
%
% See also FEM_LAME2D and AVMATRIX.

% J. Alberty, C. Carstensen, S. A. Funken, and R. Klose 07-03-00
% File <show.m> in $(HOME)/acfk/fem_lame2d/cooks/ and
% $(HOME)/acfk/fem_lame2d/lshape_p1/ and
% $(HOME)/acfk/fem_lame2d/lshape_q1/ and
% $(HOME)/acfk/fem_lame2d/hole/


%% Compute AvS
Sigma3 = zeros(size(n4e,1),4);
AreaOmega = zeros(size(c4n,1),1);
AvS = zeros(size(c4n,1),4);
for j = 1:size(n4e,1)
  area4e = computeArea4e(c4n,n4e);
  AreaOmega(n4e(j,:)) = AreaOmega(n4e(j,:)) +(area4e(j)*ones(1,3))';
  PhiGrad = [1,1,1;c4n(n4e(j,:),:)']\[zeros(1,2);eye(2)];
  U_Grad = x([1;1]*2*n4e(j,:)-[1;0]*[1,1,1])*PhiGrad;
  Sigma3(j,:) = reshape(lambda*trace(U_Grad)*eye(2) ...
      +2*mu*(U_Grad+U_Grad')/2,1,4);
  AvS(n4e(j,:),:) = AvS(n4e(j,:),:) +area4e(j)*[1;1;1]*Sigma3(j,:);
end;
AvS = AvS./(AreaOmega*[1,1,1,1]);
```

```
%% Plot the solution
for i=1:size(c4n,1)
 AvC(i)=(mu/(24*(mu+lambda)^2)+1/(8*mu))*(AvS(i,1)+...
       AvS(i,4))^2+1/(2*mu)*(AvS(2)^2-AvS(1)*AvS(4));
end
colormap(1-gray)
trisurf(n4e,factor*x(1:2:size(x,1))+c4n(:,1), ...
   factor*x(2:2:size(x,1))+c4n(:,2), ...
   zeros(size(c4n,1),1), AvC, 'facecolor','interp');
hold on
view(0,90)
hold off
colorbar('vert')

drawnow;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.71: plot/plotRT04e.m

```
function plotRT04e(c4n, n4e, p, OPTtitle4p)
%% plotRT04e plots piecewise constant approximation of Raviart-Thomas flux p
% INPUT: c4n, n4e - the triangulation
% p - coefficient of the flux with respect to broken
% RT basis functions
% OPTtitle4p - optional parameter, which defines the title
% for the flux p. The default value is
```

```
% 'RT flux plot'.

    if nargin == 4
       title4p = OPTtitle4p;
    else
       title4p = 'RT flux plot';
    end

    %% Compute components of flux
    mid4e = computeMid4e(c4n, n4e);
    u4e = p(:, 1) + p(:, 3).*mid4e(:, 1);
    v4e = p(:, 2) + p(:, 3).*mid4e(:, 2);

    %% Plot
    quiver2(mid4e(:, 1), mid4e(:, 2), u4e, v4e,...
           'n=', 0.1, 'w=', [1 1]);
    axis equal tight;
    title(title4p);
    drawnow;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.72: plot/plotTriangulation.m

```
function plotTriangulation ( c4n, n4e )
%% Draw a triangular grid into a new figure.
% plotTriangulation(c4n, n4e) draws the grid defined by c4n and n4e into
```

```
% a figure.

    % Set titles of plot and window.
    title({'Mesh plot'; [num2str(size(c4n,1)), ' nodes']});
    % This can be done with triplot but patch is _much_ faster.
    % Get the coordinates for each node of each triangle.
    X1 = c4n(n4e(:,1),1);
    Y1 = c4n(n4e(:,1),2);
    X2 = c4n(n4e(:,2),1);
    Y2 = c4n(n4e(:,2),2);
    X3 = c4n(n4e(:,3),1);
    Y3 = c4n(n4e(:,3),2);
    X = [X1'; X2'; X3'];
    Y = [Y1'; Y2'; Y3'];

    % Set the colour each triangle is filled with.
    C = 'white';

    % Draw everything, make sides blue (looks more like triplot).
    patch(X,Y,C,'EdgeColor','blue');
    axis equal tight;
    drawnow;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.73: refine/closure.m

```
function n4sRefine = closure(n4e, n4sMarked)
%% closure - Mark Reference Sides.
% closure(n4e, n4sMarked) markes the reference side of each element with
% a marked side. The reference side of an element is its first one.
%  n4e is as specified in the documentation, n4sMarked has the same
% structure as n4s.
% The output is a new list of marked sides having the same structure
% as n4s.

    %% Preliminary work.
    nrNodes = max(max(n4e));
    nrElems = size(n4e,1);
    e4n = computeE4n(n4e);

    %% Initialize the matrix of marked sides.
    % This is a symmetric matrix where entry (j,k) is 1 iff there is a
    % marked side between nodes j and k.
    marked4n = sparse(n4sMarked(:,[1 2]), n4sMarked(:,[2 1]),...
                    ones(size(n4sMarked,1),2),nrNodes,nrNodes);

    %% Execute the closure algorithm.
    for curElem = 1 : nrElems
        curNodes = n4e(curElem,:);
        % While the current element's reference side is not marked and
        % another side of the current element is marked....
        while marked4n(curNodes(1),curNodes(2)) == 0 ...
              && ( marked4n(curNodes(2),curNodes(3)) == 1 ...
                   || marked4n(curNodes(3),curNodes(1)) == 1 )
            % ... mark the reference side.
            marked4n(curNodes(1),curNodes(2)) = 1;
            marked4n(curNodes(2),curNodes(1)) = 1;
            % Marking the reference side of the current element may have
            % created a similar situation on the neighbouring element. We
            % need to handle this.
            % Tricky but true: e4n(curNodes(1),curNodes(2)) will return
            % curElem while this returns the neighbouring element number:
            neighbourElem = e4n(curNodes(2),curNodes(1));
            if neighbourElem > 0
                curNodes = n4e(neighbourElem,:);
            end
        end
    end

    %% Assemble the side list for refinement.
    marked4n = triu(marked4n);
    [row,col] = find(marked4n);
    n4sRefine = [row,col];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
```

```
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.74: refine/refineBi3GB.m

```
function [c4nNew,n4eNew,n4sDbNew,n4sNbNew] = ...
                                    refineBi3GB(c4n,n4e,n4sDb,n4sNb,n4sMarked)
%% refineBi3GB - refine using the Bisec3-Green-Blue-strategy
% refineBi3GB(c4n, n4e, n4sDb, n4sNb, n4sMarked) Refines a given mesh using
% the Bisec3-Green-Blue refinement. For details on data structures
% and refinement strategies see the documentation. Input is a mesh
% defined by c4n, n4e, n4sDb, n4sNb and marked sides given by
% n4sMarked. Output is a refined mesh defined by c4nNew, n4eNew,
% n4sDbNew and n4sNbNew.

    nrNodes = size(c4n,1);
    nrElems = size(n4e,1);

    %% Closure
    n4sRefine = closure(n4e,n4sMarked);

    %% Compute newNodes4n to find new nodes faster.
    newNodes4s = sparse(n4sRefine(:,1),n4sRefine(:,2),...
                    (1:size(n4sRefine,1))'+ nrNodes, nrNodes,nrNodes);
    newNodes4s = newNodes4s + newNodes4s';

    %% Compute coordinates of new nodes.
    mid4sRefine = (c4n(n4sRefine(:,1),:)+c4n(n4sRefine(:,2),:))/2;
    c4nNew = [c4n;mid4sRefine];

    %% bisec3 refinement
    % Count elements in new triangulation. For each refined inner side, two
    % new elements will be created.
```

```
    nrNewElems = nrElems+2*size(n4sRefine,1)-size(n4sDb,1)-size(n4sNb,1);
    n4eNew = zeros(nrNewElems,3);
    ind = 0; % index to keep track of the current element number in n4eNew
    for curElem = 1 : nrElems
        curNodes = n4e(curElem,:);
        curNewNodes = [newNodes4s(curNodes(1),curNodes(2));
                       newNodes4s(curNodes(2),curNodes(3));
                       newNodes4s(curNodes(3),curNodes(1));
                      ];
        nrNewNodes4curElem = nnz(curNewNodes);
        if nrNewNodes4curElem == 0 % no refinement
            n4eNew(ind+1,:) = curNodes;
            ind = ind+1;
        elseif nrNewNodes4curElem == 1 % green refinement
            n4eNew(ind+1:ind+2,:) = ...
                [ curNodes(3) curNodes(1) curNewNodes(1);
                  curNodes(2) curNodes(3) curNewNodes(1);
                ];
            ind = ind+2;
        elseif nrNewNodes4curElem == 2
            if curNewNodes(2) > 0 % blue right
                n4eNew(ind+1:ind+3,:) = ...
                    [ curNodes(3) curNodes(1) curNewNodes(1);
                      curNewNodes(1) curNodes(2) curNewNodes(2);
                      curNodes(3) curNewNodes(1) curNewNodes(2);
                    ];
            else % blue left
                n4eNew(ind+1:ind+3,:) = ...
                    [ curNodes(1) curNewNodes(1) curNewNodes(3);
                      curNewNodes(1) curNodes(3) curNewNodes(3);
                      curNodes(2) curNodes(3) curNewNodes(1);
                    ];
            end
            ind = ind+3;
        elseif nrNewNodes4curElem == 3 % bisec3 refinement
            n4eNew(ind+1:ind+4,:) = ...
                [ curNewNodes(1) curNodes(2) curNewNodes(2);
                  curNodes(3) curNewNodes(1) curNewNodes(2);
                  curNodes(1) curNewNodes(1) curNewNodes(3);
                  curNewNodes(1) curNodes(3) curNewNodes(3);
                ];
            ind = ind+4;
        end
    end

    %% refinement of Dirichlet boundary
    nrNewDbSides= size(intersect(sort(n4sDb,2),sort(n4sRefine,2),'rows'),1);
    n4sDbNew = zeros(nrNewDbSides,2);
    ind = 0;
    for curSide = 1 : size(n4sDb,1)
        curNodes = n4sDb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
```

```
        if curNewNodes == 0
            n4sDbNew(ind + 1,:) = curNodes;
            ind = ind + 1;
        else
            n4sDbNew(ind + 1:ind + 2,:) = ...
                [ curNodes(1) curNewNodes;
                  curNewNodes curNodes(2);
                ];
            ind = ind + 2;
        end
    end

    %% refinement of Neumann boundary
    nrNewNbSides= size(intersect(sort(n4sNb,2),sort(n4sRefine,2),'rows'),1);
    n4sNbNew = zeros(nrNewNbSides,2);
    ind = 0;
    for curSide = 1 : size(n4sNb,1)
        curNodes = n4sNb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
        if curNewNodes == 0
            n4sNbNew(ind + 1,:) = curNodes;
            ind = ind + 1;
        else
            n4sNbNew(ind + 1:ind + 2,:) = ...
                [ curNodes(1) curNewNodes;
                  curNewNodes curNodes(2);
                ];
            ind = ind + 2;
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
```

```
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.75: refine/refineBi5GB.m

```
function [c4nNew,n4eNew,n4sDbNew,n4sNbNew] = ...
                                    refineBi5GB(c4n,n4e,n4sDb,n4sNb,n4sMarked)
%% refineBi5GB - refine using the Bisec5-Green-Blue-strategy
% refineBi5GB(c4n, n4e, n4sDb, n4sNb, n4sMarked) Refines a given mesh using
% the Bisec5-Green-Blue refinement. For details on data structures
% and refinement strategies see the documentation. Input is a mesh
% defined by c4n, n4e, n4sDb, n4sNb and marked sides given by
% n4sMarked. Output is a refined mesh defined by c4nNew, n4eNew,
% n4sDbNew and n4sNbNew.

    nrNodes = size(c4n,1);
    nrElems = size(n4e,1);

    %% Closure
    n4sRefine = closure(n4e,n4sMarked);

    %% Compute newNodes4n to find new nodes faster.
    newNodes4s = sparse(n4sRefine(:,1),n4sRefine(:,2),...
                    (1:size(n4sRefine,1))'+ nrNodes, nrNodes,nrNodes);
    newNodes4s = newNodes4s + newNodes4s';

    %% Compute coordinates of new nodes.
    mid4sRefine = (c4n(n4sRefine(:,1),:)+c4n(n4sRefine(:,2),:))/2;
    c4nNew = [c4n;mid4sRefine];

    %% bisec5 refinement
    % Allocate memory for new elements and additional nodes. To do this,
    % find out how many new elements will be created.
    nrNewElems = 0;
    nrNewNodes = 0;
    for curElem = 1: nrElems
        curNodes = n4e(curElem,:);
        curNewNodes = [newNodes4s(curNodes(1),curNodes(2));
                       newNodes4s(curNodes(2),curNodes(3));
                       newNodes4s(curNodes(3),curNodes(1));
                      ];
        nrNewNodes4curElem = nnz(curNewNodes);
        nrNewElems = nrNewElems + nrNewNodes4curElem;
        if nrNewNodes4curElem == 3 % This element will be bisec5'd.
            nrNewElems = nrNewElems + 2; % Need 2 more elements and...
            nrNewNodes = nrNewNodes + 1; % ... 1 more node.
        end
    end
    n4eNew = zeros(nrElems+nrNewElems,3);
    n4eInd = 0;
    c4nInd = size(c4nNew,1);
```

```
    c4nNew = [c4nNew;zeros(nrNewNodes,2)];
    % Start the refinement.
    for curElem = 1 : nrElems
        curNodes = n4e(curElem,:);
        curNewNodes = [newNodes4s(curNodes(1),curNodes(2));
                       newNodes4s(curNodes(2),curNodes(3));
                       newNodes4s(curNodes(3),curNodes(1));
                      ];
        nrNewNodes4curElem = nnz(curNewNodes);
        if nrNewNodes4curElem == 0 % no refinement
            n4eNew(n4eInd+1,:) = curNodes;
            n4eInd = n4eInd + 1;
        elseif nrNewNodes4curElem == 1 % green refinement
            n4eNew(n4eInd+1:n4eInd+2,:) = ...
                [ curNodes(3) curNodes(1) curNewNodes(1);
                  curNodes(2) curNodes(3) curNewNodes(1);
                ];
            n4eInd = n4eInd+2;
        elseif nrNewNodes4curElem == 2
            if curNewNodes(2) > 0 % blue right
                n4eNew(n4eInd+1:n4eInd+3,:) = ...
                    [ curNodes(3) curNodes(1) curNewNodes(1);
                      curNewNodes(1) curNodes(2) curNewNodes(2);
                      curNodes(3) curNewNodes(1) curNewNodes(2);
                    ];
            else % blue left
                n4eNew(n4eInd+1:n4eInd+3,:) = ...
                    [ curNodes(1) curNewNodes(1) curNewNodes(3);
                      curNewNodes(1) curNodes(3) curNewNodes(3);
                      curNodes(2) curNodes(3) curNewNodes(1);
                    ];
            end
            n4eInd = n4eInd+3;
        elseif nrNewNodes4curElem == 3
            % bisec5 refinement
            c4nNew(c4nInd+1,:) = (c4n(curNodes(3),:)+...
                                  c4nNew(curNewNodes(1),:))./2;
            c4nInd = c4nInd + 1;
            n4eNew(n4eInd+1:n4eInd+6,:) = ...
                [ curNewNodes(1) curNodes(2) curNewNodes(2);
                  curNodes(1) curNewNodes(1) curNewNodes(3);
                  curNewNodes(1) curNewNodes(2) c4nInd;
                  curNewNodes(2) curNodes(3) c4nInd;
                  curNewNodes(3) curNewNodes(1) c4nInd;
                  curNodes(3) curNewNodes(3) c4nInd;
                ];
            n4eInd = n4eInd + 6;
        end
    end

    %% refinement of Dirichlet boundary
    nrNewDbSides= size(intersect(sort(n4sDb,2),sort(n4sRefine,2),'rows'),1);
```

```
    n4sDbNew = zeros(nrNewDbSides,2);
    ind = 0;
    for curSide = 1 : size(n4sDb,1)
        curNodes = n4sDb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
        if curNewNodes == 0
            n4sDbNew(ind + 1,:) = curNodes;
            ind = ind + 1;
        else
            n4sDbNew(ind + 1:ind + 2,:) = ...
                [ curNodes(1) curNewNodes;
                  curNewNodes curNodes(2)
                ];
            ind = ind + 2;
        end
    end

    %% refinement of Neumann boundary
    nrNewNbSides= size(intersect(sort(n4sNb,2),sort(n4sRefine,2),'rows'),1);
    n4sNbNew = zeros(nrNewNbSides,2);
    ind = 0;
    for curSide = 1 : size(n4sNb,1)
        curNodes = n4sNb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
        if curNewNodes == 0
            n4sNbNew(ind + 1,:) = curNodes;
            ind = ind + 1;
        else
            n4sNbNew(ind + 1:ind + 2,:) = ...
                [ curNodes(1) curNewNodes;
                  curNewNodes curNodes(2)
                ];
            ind = ind + 2;
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
```

```
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.76: refine/refineRGB.m

```
function [c4n,n4e,n4sDb,n4sNb,Pe4e,Pn4n,n4ms] ...
    = refineRGB(c4n,n4e,n4sDb,n4sNb,ms)
%% refineRGB
% [c4n,n4e,n4sDb,n4sNb,Pe4e,Pn4n]
% = refineRGB(c4n,n4e,n4sDb,n4sNb,ms)
%
% Refine marked sides of a grid ("red/green/blue").
% Marked sides are given with their node numbers in n4ms
% (a [#Marked-sides 2] vector with n4s-structure).
% A closure algorithm is applied to n4ms first to ensure that
% the reference sides of all marked elements are marked.
%
% ms can be a simple list ([? x 1]!!) of number or marked
% (numbering as in n4s), or a list of node pairs ([? x 2]),
% i.e., a (row-)subset of n4s.
%
% Pn4n is a [#nodes-in-new-grid 2] matrix where row j contains
% the number of the two (old) nodes that touch the (old) side
% on which the (new) node j is placed. If j is the number of
% an old node, the row simply contains [j j]. This allows for
% easy prolongation of conforming P1 functions which are
% represented by a [#nodes] vector:
% u = mean(u(Pn4n),2)
%
% Pe4e is a [#elemenents-in-new-grid] vector where entry j
% contains the number of the (old) element that covers the
% (new) element j. This allows for easy prolongation of
% P0 functions which are represented by a [#elements] vector:
% u = u(Pe4e);
%
% mostly (C) 2009--2012 W. Boiger, HU Berlin
% Licensed under GNU GPL 3+. No warranty! See LICENSE.txt.
% Modified (C. Merdon): improved and some fixes

n4s = computeN4s(n4e);
s4n = computeS4n(n4e,n4s);
s4e = computeS4e(n4e);

if size(ms,2)==2 % ms is really n4ms ==> transform it
    ms = SidesFromN4s(ms,s4n); % Unclosed marked sides list
end
```

```
ms = closure(ms,s4e); % (Closed) list of marked sides
n4ms = n4s(ms,:);
mid4ms = computeMid4s(c4n,n4ms);
% s4ms(k)==j : k-th marked side is side j (of all sides)
s4ms = SidesFromN4s(n4ms,s4n);
% nNew4s(k)==j>0 : Old side k is marked and, its center will
% be the new node j
% nNew4s(k)==0 : Old side k is not marked and remains
% New nodes are simply attached to the old c4n in the order
% given by ms (==> n4ms ==> mid4ms)
nNew4s = zeros(size(n4s,1),1);
nNew4s(s4ms) = (1:length(s4ms))+size(c4n,1);
Pn4n = [(1:size(c4n,1))'*[1 1];n4ms];
c4n = [c4n;mid4ms]; % ** Update c4n with new nodes
% nNew4e(k,m)==j>0 : Side m (1,2 or 3) of (old) element k is
% marked and will generate the new node j
% nNew4e(k,m)==0 : Side m of (old) element k is not marked
nNew4e = reshape(nNew4s(s4e),[],size(s4e,2));
% ** Core refinement algorithm
% Get lists of numbers of (old) elements:
% e0 : Not to be refined (all sides remain untouched)
% er : Red (all sides are marked, thus to be refined)
% eg : Green (only reference side is marked)
% eb/eB : Blue (reference and 2nd/3rd side are marked)
e0 = find(~any(nNew4e,2));
er = find(all(nNew4e,2));
eg = find(and(nNew4e(:,1),~any(nNew4e(:,[2 3]),2)));
eb = find(and(all(nNew4e(:,[1 2]),2),~nNew4e(:,3)));
eB = find(and(all(nNew4e(:,[1 3]),2),~nNew4e(:,2)));
% n3 Element [n1 n2 n3] (as of a row in n4e)
% / \ has the sides/new nodes [s1 s2 s3] (as
% s3 s2 given by a row in nNew4e), in the order
% / \ depicted here. Each sj can be a new
% n1 --s1-- n2 node, depending of the refinement type.
n4e = [ n4e(e0,:) % Untouched
    n4e(er,1) nNew4e(er,1) nNew4e(er,3) % Red
    nNew4e(er,1) n4e(er,2) nNew4e(er,2)
    nNew4e(er,3) nNew4e(er,2) n4e(er,3)
    nNew4e(er,[2 3 1])
    n4e(eg,3) n4e(eg,1) nNew4e(eg,1) % Green
    n4e(eg,2) n4e(eg,3) nNew4e(eg,1)
    n4e(eb,3) n4e(eb,1) nNew4e(eb,1) % Blue b
    nNew4e(eb,1) n4e(eb,2) nNew4e(eb,2)
    n4e(eb,3) nNew4e(eb,1) nNew4e(eb,2)
    n4e(eB,2) n4e(eB,3) nNew4e(eB,1) % Blue B
    nNew4e(eB,1) n4e(eB,3) nNew4e(eB,3)
    n4e(eB,1) nNew4e(eB,1) nNew4e(eB,3) ];
Pe4e = [ repmat(e0,1,1)
    repmat(er,4,1)
    repmat(eg,2,1)
    repmat(eb,3,1)
    repmat(eB,3,1) ];
```

```
clear('e0','er','eg','eb','eB');
% ** Refine boundary
n4sDb = refineBoundary(n4sDb,s4n,nNew4s);
n4sNb = refineBoundary(n4sNb,s4n,nNew4s);
end

function ms = closure(ms,s4e)
% Append more sides to list of marked sides (ms) until the
% reference sides of all elements that have at least one
% marked side are marked.
m4s = zeros(max(s4e(:)),1);
m4s(ms) = 1; % m4s(k)==1 if side k marked, ==0 otherwise
while true
    todo4e = reshape(m4s(s4e),[],size(s4e,2));
    todo4e = and(any(todo4e(:,[2 3]),2),~todo4e(:,1));
    % Now: todo4e(k)==1 : Element k has a marked side, but its
    % reference side is not marked, so we need to mark it
    % also, and then check again.
    if ~any(todo4e) % If all elements are consistent => done!
        break;
    end
    sTodo = s4e(find(todo4e),1); % List of sides to be marked
    sTodo = unique(sTodo);
    m4s(sTodo) = 1; % Mark those sides
end
ms = find(m4s);
end

function n4sB = refineBoundary(n4sB,s4n,nNew4s)
% Refine the boundary given by it nodes in n4sB
% s4n must be the full s4n matrix of the old grid.
if isempty(n4sB)
    return
end
% Get new (middel) node for each boundary side
nNew4sB = SidesFromN4s(n4sB,s4n);
nNew4sB = nNew4s(nNew4sB)';
%nNew4sB = reshape(nNew4sB,1,[]); % TODO Is this needed
% nNew4sB(j)=k>0 : Boundary side j will become new node k
% =k=0 : Boundary side j won't be modified
n4sB = n4sB';
n4sB = [n4sB(1,:) ; nNew4sB ; nNew4sB ; n4sB(2,:)];
n4sB = reshape(n4sB,[],1);
n4sB = n4sB(find(n4sB));
n4sB = reshape(n4sB,2,[]);
n4sB = n4sB';
end

function s = SidesFromN4s(n4s,s4n)
% Get list of sides from n4s structure
% s4n must refer to full grid!
% (This is a copy from refineUniformRed.m)
```

```
s = s4n(n4s(:,1),n4s(:,2));
s = diag(s);
s = full(s);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.77: refine/refineUniformRed.m

```
function [c4nNew,n4eNew,n4sDbNew,n4sNbNew] = refineUniformRed(c4n,n4e,n4sDb,n4sNb)
%% refineUniformRed - Refine every element the "red" way.
% refineUniformRed(c4n, n4e, n4sDb, n4sNb) Refines a given mesh uniformly
% using the red refinement. For details on data structures and
% refinement strategies see the documentation. Input is a mesh
% defined by c4n, n4e, n4sDb and n4sNb. Output is a refined mesh
% defined by c4nNew, n4eNew, n4sDbNew and n4sNbNew.

    %% Preliminary work.
    nrNodes = size(c4n,1);
    nrElems = size(n4e,1);
    n4s = computeN4s(n4e);
    nrSides = size(n4s,1);
    newNodes4s = sparse(n4s(:,1),n4s(:,2),(1:nrSides)'+ nrNodes, ...
                        nrNodes,nrNodes);
    newNodes4s = newNodes4s + newNodes4s';

    %% Compute coordinates for new nodes.
    % As every element is refined "red", there is a new node on each side.
    mid4s = computeMid4s(c4n,n4s);
    c4nNew = [c4n;mid4s];

```

```
    %% red refinement
    n4eNew = zeros(4*nrElems,3);
    for curElem = 1 : nrElems
        curNodes = n4e(curElem,:);
        curNewNodes = [newNodes4s(curNodes(1),curNodes(2));
                       newNodes4s(curNodes(2),curNodes(3));
                       newNodes4s(curNodes(3),curNodes(1));
                      ];
        % Generate new elements.
        n4eNew(4*(curElem-1)+1:4*curElem,:) = ...
                 [ curNodes(1) curNewNodes(1) curNewNodes(3);
                   curNewNodes(1) curNodes(2) curNewNodes(2);
                   curNewNodes(2) curNewNodes(3) curNewNodes(1);
                   curNewNodes(3) curNewNodes(2) curNodes(3);
                 ];
    end

    %% refinement of Dirichlet boundary
    n4sDbNew = zeros(2*size(n4sDb,1),2);
    for curSide = 1 : size(n4sDb,1)
        curNodes = n4sDb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
        % Generate new Dirichlet boundary sides.
        n4sDbNew(2*(curSide-1)+1:2*curSide,:) = ...
                 [ curNodes(1) curNewNodes;
                   curNewNodes curNodes(2);
                 ];
    end

    %% refinement of Neumann boundary
    n4sNbNew = zeros(2*size(n4sNb,1),2);
    for curSide = 1 : size(n4sNb,1)
        curNodes = n4sNb(curSide,:);
        curNewNodes = newNodes4s(curNodes(1),curNodes(2));
        % Generate new Neumann boundary sides.
        n4sNbNew(2*(curSide-1)+1:2*curSide,:) = ...
                 [ curNodes(1) curNewNodes;
                   curNewNodes curNodes(2);
                 ];
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
```

```
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.78: solve/solveAWElasticity.m

```
function [x,u,p,energy,A,sigma,displacement,divSigma] = solveAWElasticity(c4n,n4e,n4sDb,n4sNb,lambda,mu,

    nrElems = size(n4e,1);
    area4e = getArea4e(c4n,n4e);
    [c21,c31] = transformation(c4n,n4e);
    refgradP1 = [-1 -1; 1 0; 0 1];

    basisCoefficients = getBasisCoefficients(c4n,n4e,c21,c31);

    [dofU4e,dofSigma4e,nrDoFU,nrDoFSigma] = getDofs(c4n,n4e);


    localM = 1/2520 * ...
     [ 420 210 210 84 42 84 14 0 -14 14
       210 420 210 84 84 42 -14 14 0 14
       210 210 420 42 84 84 0 -14 14 14
        84 84 42 28 14 14 0 2 -2 4
        42 84 84 14 28 14 -2 0 2 4
        84 42 84 14 14 28 2 -2 0 4
        14 -14 0 0 -2 2 3 -1 -1 0
         0 14 -14 2 0 -2 -1 3 -1 0
       -14 0 14 -2 2 0 -1 -1 3 0
        14 14 14 4 4 4 0 0 0 1 ];

    localN1 = 1/12*[ones(3,3)+eye(3,3), zeros(3,3); zeros(3,3), ones(3,3)+eye(3,3)];
    localN2 = eye(6,6);
    localN = localN1*localN2(:,[1 4 2 5 3 6]);

    C11 = (lambda + 2*mu)/(4*mu*(lambda+mu));
    C12 = -lambda /(4*mu*(lambda+mu));
    C33 = 1/(mu);

    A = zeros(30,30,nrElems);

    for curElem = 1 : nrElems
        % local A
```

```
        a = permute(basisCoefficients(curElem,:,1:3:end),[2 3 1])';
        b = permute(basisCoefficients(curElem,:,2:3:end),[2 3 1])';
        c = permute(basisCoefficients(curElem,:,3:3:end),[2 3 1])';
        localA = area4e(curElem)*(...
                        C11*(a'*localM*a + b'*localM*b) + ...
                        C12*(a'*localM*b + b'*localM*a) + ...
                        C33*(c'*localM*c) );
        % local B
        gradP1 = ([c31(curElem,2),-c21(curElem,2);-c31(curElem,1),c21(curElem,1)]*refgradP1')';
        L1 = [gradP1(1,1),0,gradP1(1,2);0,gradP1(1,2),gradP1(1,1)];
        L2 = [gradP1(2,1),0,gradP1(2,2);0,gradP1(2,2),gradP1(2,1)];
        L3 = [gradP1(3,1),0,gradP1(3,2);0,gradP1(3,2),gradP1(3,1)];
        O = zeros(2,3);
        D = [L1, L2, L3, L2, O, L3, L2, O, -L3, O;
             L1, L2, L3, L1, L3, O,-L1, L3, O, O;
             L1, L2, L3, O, L2, L1, O, -L2, L1, O];
        C = permute(basisCoefficients(curElem,:,:),[2 3 1]);
        D = D*C';
        localB = area4e(curElem)*localN*D;
        localB = localB*sqrt(area4e(curElem));
        % local matrix
        A(:,:,curElem) = [localA, localB';
                          localB, zeros(6,6)];
    end

    %Assembly of global Matrix
    dof4e = [dofSigma4e,dofU4e];
    [I,J] = localDoFtoGlobalDoF(dof4e);
    A = sparse(I,J,A(:));

    % right hand side bD
    u4DbVal = integrate(c4n,n4Db,@(n4p,curGpt4p,Gpts4ref)intU4Db(n4p,curGpt4p,Gpts4ref,c4n,u4Db
    %u4DbVal = integrateQUADV(c4n,n4Db,@(n4p,curGpt4p,Gpts4ref)intU4Db(n4p,curGpt4p,Gpts4ref,c4
    dof = dofSigma4e(s4ed(s4Db,1),:);
    b = accumarray(dof(:),u4DbVal(:),[size(A,2) 1]);

    % right hand side bf
    fval = -integrate(c4n,n4e,@(n4p,curGpt4p,Gpts4ref)intf(n4p,curGpt4p,Gpts4ref,f),4);
    fval = fval.*(sqrt(area4e)*ones(1,6));
    b = b + accumarray(dofU4e(:),fval(:),[size(A,2) 1]);

    % Neumann bd conditions
    x = zeros(size(A,2),1);
    fixNodes = [];
    B = [];
    bN = [];
    if ~isempty(mesh.n4Nb)
        % moments
        moments0 = (1./mesh.length4ed(mesh.ed4Nb)*ones(1,2)).*integrate(@(points,refp,ind)intMom
        moments1 = (1./mesh.length4ed(mesh.ed4Nb)).^2*ones(1,2).*integrate(@(points,refp,ind)int
        ed4Nb = mesh.ed4Nb;
        fixNodes = 3*mesh.nrNodes+[ed4Nb,mesh.nrEdges+ed4Nb,2*mesh.nrEdges+ed4Nb,3*mesh.nrEdges+
```

```
        moments = [moments0,moments1];
        x(fixNodes(:)) = moments(:);
        fixNodes = fixNodes(:);
        % nodes
        NbNodes = unique(mesh.n4Nb(:));
        NbNodesDof = zeros(mesh.nrNodes,2);
        NbNodesDof(NbNodes,:) = reshape(1:2*length(NbNodes),[],2);
        B = sparse(2*length(NbNodes),size(A,2));
        bN = zeros(2*length(NbNodes),1);
        fixNbNodes = [];
        NbEd4n = mesh.NbEd4n;
        normal4ed = mesh.normal4ed;
        for j = 1 : length(NbNodes)
            curNbEdges = NbEd4n(NbNodes(j),:);
            % one Nb one Db
            if nnz(curNbEdges) < 2
                if curNbEdges(2) == 0
                    curNormal = normal4ed(curNbEdges(1),:);
                else
                    curNormal = normal4ed(curNbEdges(2),:);
                end
                bN(NbNodesDof(NbNodes(j),:)) = g(mesh.c4n(NbNodes(j),:),curNormal);
                B(NbNodesDof(NbNodes(j),1),NbNodes(j)) = curNormal(1);
                B(NbNodesDof(NbNodes(j),2),mesh.nrNodes+NbNodes(j)) = curNormal(2);
                B(NbNodesDof(NbNodes(j),:),2*mesh.nrNodes+NbNodes(j)) = curNormal([2 1]);
            else % two adjacent Nb Edges
                curNormals = normal4ed(curNbEdges,:);
                % Parallel
                if sum(curNormals(1,:).*curNormals(2,:)) == 1
                    curNormal = curNormals(1,:);
                    bN(NbNodesDof(NbNodes(j),:)) = g(mesh.c4n(NbNodes(j),:),curNormal);
                    B(NbNodesDof(NbNodes(j),1),NbNodes(j)) = curNormal(1);
                    B(NbNodesDof(NbNodes(j),2),mesh.nrNodes+NbNodes(j)) = curNormal(2);
                    B(NbNodesDof(NbNodes(j),:),2*mesh.nrNodes+NbNodes(j)) = curNormal([2 1]);
                else % non parrallel
                    curG = g([mesh.c4n(NbNodes(j),:);mesh.c4n(NbNodes(j),:)],curNormals)';
                    curA = [curNormals(1,1),0,curNormals(1,2);
                        0, curNormals(1,2), curNormals(1,1);
                        curNormals(2,1),0,curNormals(2,2);
                        0, curNormals(2,2), curNormals(2,1)];
                    z = pinv(curA)*curG(:);
                    dofSigma = [NbNodes(j),mesh.nrNodes+NbNodes(j),2*mesh.nrNodes+NbNodes(j)];
                    x(dofSigma) = z;
                    fixNbNodes = [fixNbNodes,NbNodesDof(NbNodes(j),:)];
                    fixNodes = [fixNodes;dofSigma(:)];
                end
            end
        end
        freeNbNodes = setdiff(1:length(bN),fixNbNodes);
        B = B(freeNbNodes,:);
        bN = bN(freeNbNodes,:);
    end
```

```
    % Solve
    dofSigma = 1:aw.nrDoFSigma;
    dofU = aw.nrDoFSigma+1 : aw.nrDoFSigma+aw.nrDoFU;
    b = b - [A(dofSigma,dofSigma)*x(dofSigma);
             A(dofU,dofSigma)*x(dofSigma)];
    freeNodes = setdiff(1:size(A,2),fixNodes(:));
    if ~isempty(B)
        B = B(:,freeNodes);
    end
    x([freeNodes,aw.nrDoFSigma+aw.nrDoFU+1:aw.nrDoFSigma+aw.nrDoFU+length(bN)]) = ...
        [A(freeNodes,freeNodes),B';
         B,zeros(length(bN),length(bN))] \ [b(freeNodes);bN];
    x = x(1:aw.nrDoFSigma+aw.nrDoFU);
    x(aw.dofU4e) = x(aw.dofU4e).*(sqrt(mesh.area4e)*ones(1,size(aw.dofU4e,2)));

    u = x(dofU4e);
    p = x(dofSigma4e);

    energy = x(dofSigma)'*A(dofSigma,dofSigma)*x(dofSigma);
    A = [A(freeNodes,freeNodes),B';
         B,zeros(length(bN),length(bN))];
end

function val = intf(n4p,curGpt4p,Gpts4ref,f)
    fval = f(curGpt4p);
    p1Basis = [1 - Gpts4ref(1) - Gpts4ref(2), Gpts4ref(1), Gpts4ref(2)];
    p1Basis = (p1Basis(:)*ones(1,size(n4p,1)))';
    val = [fval(:,1)*ones(1,3),fval(:,2)*ones(1,3)].*[p1Basis,p1Basis];
end

function val = intU4Db(n4p,curGpt4p,Gpts4ref,c4n,u4Db,n4e,basisCoefficients,n4sDb)
    if isempty(curGpt4p)
        c1 = c4n(n4p(:,1),:);
        c2 = c4n(n4p(:,2),:);
        curGpt4p = c1 + Gpts4ref(1,1)*(c2-c1);
    end
    u4DbVal = u4Db(curGpt4p);
    normals = computeNormal4s(c4n,n4p);
    Gpts4ref = [Gpts4ref(1),0;1-Gpts4ref(1),Gpts4ref(1);0,1-Gpts4ref(1)];
    basisSigma4eEd1 = basisSigma(Gpts4ref(1,1),Gpts4ref(1,2),basisCoefficients,n4e);
    basisSigma4eEd2 = basisSigma(Gpts4ref(2,1),Gpts4ref(2,2),basisCoefficients,n4e);
    basisSigma4eEd3 = basisSigma(Gpts4ref(3,1),Gpts4ref(3,2),basisCoefficients,n4e);
    sigma = zeros([size(n4p,1) size(basisSigma4eEd1,2) size(basisSigma4eEd1,3)]);
    edNumber = computeEdNumber(e4s,s4e);
    elements = rowaddr(e4s,n4sDb(:,1),n4sDb(:,2));
    ind1 = find(edNumber==1);
    ind2 = find(edNumber==2);
    ind3 = find(edNumber==3);
    if ~isempty(ind1); sigma(ind1,:,:) = basisSigma4eEd1(elements(ind1),:,:);end
    if ~isempty(ind2); sigma(ind2,:,:) = basisSigma4eEd2(elements(ind2),:,:);end
    if ~isempty(ind3); sigma(ind3,:,:) = basisSigma4eEd3(elements(ind3),:,:);end
```

```
    u4DbVal1 = u4DbVal(:,1)*ones(1,size(sigma,3));
    u4DbVal2 = u4DbVal(:,2)*ones(1,size(sigma,3));
    normals1 = normals(:,1)*ones(1,size(sigma,3));
    normals2 = normals(:,2)*ones(1,size(sigma,3));
    sigma = permute(sigma,[1 3 2]);
    val = u4DbVal1.*(sigma(:,:,1).*normals1 + sigma(:,:,3).*normals2) +...
          u4DbVal2.*(sigma(:,:,3).*normals1 + sigma(:,:,2).*normals2);
end

function val = intMoment0(points,refPoints,indices,g,mesh)
    normals = mesh.normal4ed(mesh.ed4Nb,:);
    val = g(points,normals);
end

function val = intMoment1(points,refPoints,indices,g,mesh)
tangents = mesh.tangent4ed(mesh.ed4Nb,:);
normals = mesh.normal4ed(mesh.ed4Nb,:);
mid = mesh.mid4ed(mesh.ed4Nb,:);
val = (sum((points-mid).*tangents,2)*ones(1,2)).*g(points,normals);
end

function basisCoefficients = getBasisCoefficients(c4n,n4e,c21,c31)
normals4e = obj.mesh.normal4e;
nrElems = obj.mesh.nrElems;
[c21,c31] = obj.mesh.transformation;
ed4e = obj.mesh.ed4e;
normal4ed = obj.mesh.normal4ed;
length4ed = obj.mesh.length4ed;
% Initialisation
obj.basisCoefficients = zeros(nrElems,24,30);
C = zeros(30,30);
% Nodal degrees of freedom
C(1:9,1:9) = 60*eye(9,9);
% Element degrees of freedom
I = [eye(3,3),eye(3,3),eye(3,3)];
C(22:24,:) = [20*I,5*I,0*I,eye(3,3)];
% Element specific transformation
for curElem = 1 : nrElems
    % Edge degree of
    curNormal = normal4ed(ed4e(curElem,:),:);
    N1 = [diag(curNormal(1,:)),curNormal(1,[2 1])'];
    N2 = [diag(curNormal(2,:)),curNormal(2,[2 1])'];
    N3 = [diag(curNormal(3,:)),curNormal(3,[2 1])'];
    curNormal = permute(normals4e(curElem,:,:),[2 3 1]);
    M1 = [diag(curNormal(1,:)),curNormal(1,[2 1])'];
    M2 = [diag(curNormal(2,:)),curNormal(2,[2 1])'];
    M3 = [diag(curNormal(3,:)),curNormal(3,[2 1])'];
    O = zeros(2,3);
    R = [30*N1,30*N1,O;-5*M1,5*M1,O;
        O,30*N2,30*N2;O,-5*M2,5*M2;
        30*N3,O,30*N3;5*M3,O,-5*M3];
    S1 = [10*N1,O,O;O,O,O;
```

```
        0,10*N2,0;0,0,0;
        0,0,10*N3;0,0,0];
    S2 = [0,0,0;-M1,0,0;
        0,0,0;0,-M2,0;
        0,0,0;0,0,-M3];
    C(10:21,1:27) = [R,S1,S2];
    % Constraint div(sigma) in P1
    gradP1 = obj.gradP1(1,1);
    gradP1 = permute(gradP1(1,:,:),[2 3 1]);
    gradP1 = [c31(curElem,2),-c21(curElem,2);-c31(curElem,1),c21(curElem,1)]*gradP1';
    gradP1X = gradP1(1,:);
    gradP1Y = gradP1(2,:);
    dxW = [3*(gradP1X(1)-gradP1X(2)), -gradP1X(3), gradP1X(3), gradP1X(3);
        gradP1X(1), 3*(gradP1X(2)-gradP1X(3)), -gradP1X(1), gradP1X(1);
        -gradP1X(2), gradP1X(2), 3*(gradP1X(3)-gradP1X(1)), gradP1X(2)];
    dyW = [3*(gradP1Y(1)-gradP1Y(2)), -gradP1Y(3), gradP1Y(3), gradP1Y(3);
        gradP1Y(1), 3*(gradP1Y(2)-gradP1Y(3)), -gradP1Y(1), gradP1Y(1);
        -gradP1Y(2), gradP1Y(2), 3*(gradP1Y(3)-gradP1Y(1)), gradP1Y(2)];
    0 = zeros(3,1);
    Q = [dxW(:,1),0,dyW(:,1),dxW(:,2),0,dyW(:,2),dxW(:,3),0,dyW(:,3),dxW(:,4),0,dyW(:,4);
        0,dyW(:,1),dxW(:,1),0,dyW(:,2),dxW(:,2),0,dyW(:,3),dxW(:,3),0,dyW(:,4),dxW(:,4)];
    C(25:30,19:30) = 60*max(length4ed(ed4e(curElem,:)))*Q;
    % Assembly
    x = (C/60)\[eye(24);zeros(6,24)];
    basisCoefficients(curElem,:,:) = x';
end
end

%% Organisation of the degrees of freedom
function [dofU4e,dofSigma4e,nrDoFU,nrDoFSigma] = getDofs(c4n,n4e)
n4e = obj.mesh.n4e;
nrNodes = obj.mesh.nrNodes;
n4s = computeN4s(c4n,n4e);
nrEdges = size(n4s,1);
nrElems = obj.mesh.nrElems;
% Stress field degrees of freedom (24)
dofSigma4e = zeros(nrElems,24);
% nodal degrees of freedom (z1,0,0), (0,z1,0), (0,0,z1),...
dofSigma4e(:,1:9) = [n4e,nrNodes+n4e,2*nrNodes+n4e];
dofSigma4e(:,1:9) = obj.dofSigma4e(:,[1 4 7 2 5 8 3 6 9]);
% edge degrees of freedom
ed4e = 3*nrNodes + obj.mesh.ed4e;
dofSigma4e(:,10:21) = [ed4e,nrEdges+ed4e,2*nrEdges+ed4e,3*nrEdges+ed4e];
dofSigma4e(:,10:21) = obj.dofSigma4e(:,[10 13 16 19 11 14 17 20 12 15 18 21]);
% element degrees of freedom
elems = 3*nrNodes + 4*nrEdges + (1:nrElems)';
dofSigma4e(:,22:24) = [elems, nrElems+elems, 2*nrElems+elems];
% Displacement degrees of freedom (6)
% (z1;0),(z2;0),(z3;0),(0;z1),(0;z2),(0;z3)
elems = 3*nrNodes + 4*nrEdges + 3*nrElems + (1:nrElems)';
dofU4e = [elems, nrElems+elems, 2*nrElems+elems,...
    3*nrElems+elems, 4*nrElems+elems,5*nrElems+elems];
```

```
% nrDoFs
nrDoFSigma = 3*nrNodes + 4*nrEdges + 3*nrElems;
nrDoFU = 6*nrElems;
end

%% P3 basis functions
function val = basisP3(x,y)
basisP1 = obj.basisP1(x,y);
val = zeros(size(x,1),10);
val(:, 1) = basisP1(:,1);
val(:, 2) = basisP1(:,2);
val(:, 3) = basisP1(:,3);
val(:, 4) = basisP1(:,1).*basisP1(:,2);
val(:, 5) = basisP1(:,2).*basisP1(:,3);
val(:, 6) = basisP1(:,3).*basisP1(:,1);
val(:, 7) = basisP1(:,1).*basisP1(:,2).*(basisP1(:,1) - basisP1(:,2));
val(:, 8) = basisP1(:,2).*basisP1(:,3).*(basisP1(:,2) - basisP1(:,3));
val(:, 9) = basisP1(:,3).*basisP1(:,1).*(basisP1(:,3) - basisP1(:,1));
val(:,10) = basisP1(:,1).*basisP1(:,2).*basisP1(:,3);
end

function val = gradP3(x,y)
gradP1 = obj.gradP1(x,y);
basisP1 = obj.basisP1(x,y);

val = zeros(10,2);
val(1,:) = gradP1(:,1,:);
val(2,:) = gradP1(:,2,:);
val(3,:) = gradP1(:,3,:);
val(4,:) = gradP1(:,1,:)*basisP1(:,2) + gradP1(:,2,:)*basisP1(:,1);
val(5,:) = gradP1(:,2,:)*basisP1(:,3) + gradP1(:,3,:)*basisP1(:,2);
val(6,:) = gradP1(:,3,:)*basisP1(:,1) + gradP1(:,1,:)*basisP1(:,3);
val(7,:) = gradP1(:,1,:)*basisP1(:,2)*(basisP1(:,1) - basisP1(:,2)) + ...
    basisP1(:,1)*gradP1(:,2,:)*(basisP1(:,1) - basisP1(:,2)) + ...
    basisP1(:,1)*basisP1(:,2)*(gradP1(:,1,:) - gradP1(:,2,:));
val(8,:) = gradP1(:,2,:)*basisP1(:,3)*(basisP1(:,2) - basisP1(:,3)) + ...
    basisP1(:,2)*gradP1(:,3,:)*(basisP1(:,2) - basisP1(:,3)) + ...
    basisP1(:,2)*basisP1(:,3)*(gradP1(:,2,:) - gradP1(:,3,:));
val(9,:) = gradP1(:,3,:)*basisP1(:,1)*(basisP1(:,3) - basisP1(:,1)) + ...
    basisP1(:,3)*gradP1(:,1,:)*(basisP1(:,3) - basisP1(:,1)) + ...
    basisP1(:,3)*basisP1(:,1)*(gradP1(:,3,:) - gradP1(:,1,:));
val(10,:) = gradP1(:,1,:)*basisP1(:,2)*basisP1(:,3) + ...
    basisP1(:,1)*gradP1(:,2,:)*basisP1(:,3) + ...
    basisP1(:,1)*basisP1(:,2)*gradP1(:,3,:);
end

function val = d2P3(x,y)
gradP1 = [-1 -1 ; 1 0; 0 1];
basisP1 = obj.basisP1(x,y);

val = zeros(10,3);
val(4,:) = [2*gradP1(1,1)*gradP1(2,1),2*gradP1(1,2)*gradP1(2,2),gradP1(1,1)*gradP1(2,2)+gradP1(1,2)*gradP
```

```
val(5,:) = [2*gradP1(2,1)*gradP1(3,1),2*gradP1(2,2)*gradP1(3,2),gradP1(2,1)*gradP1(3,2)+gradP1(
val(6,:) = [2*gradP1(3,1)*gradP1(1,1),2*gradP1(3,2)*gradP1(1,2),gradP1(3,1)*gradP1(1,2)+gradP1(
val(7,:) = [2*gradP1(1,1)*gradP1(2,1)*(basisP1(1)-basisP1(2)) + 2*basisP1(1)*gradP1(2,1)*(gradP
    2*gradP1(1,2)*gradP1(2,2)*(basisP1(1)-basisP1(2)) + 2*basisP1(1)*gradP1(2,2)*(gradP1(1,2)-g
    gradP1(1,1)*gradP1(2,2)*(basisP1(1)-basisP1(2)) + gradP1(1,1)*basisP1(2)*(gradP1(1,2)-gradP
val(8,:) = [2*gradP1(2,1)*gradP1(3,1)*(basisP1(2)-basisP1(3)) + 2*basisP1(2)*gradP1(3,1)*(gradP
    2*gradP1(2,2)*gradP1(3,2)*(basisP1(2)-basisP1(3)) + 2*basisP1(2)*gradP1(3,2)*(gradP1(2,2)-g
    gradP1(2,1)*gradP1(3,2)*(basisP1(2)-basisP1(3)) + gradP1(2,1)*basisP1(3)*(gradP1(2,2)-gradP
val(9,:) = [2*gradP1(3,1)*gradP1(1,1)*(basisP1(3)-basisP1(1)) + 2*basisP1(3)*gradP1(1,1)*(gradP
    2*gradP1(3,2)*gradP1(1,2)*(basisP1(3)-basisP1(1)) + 2*basisP1(3)*gradP1(1,2)*(gradP1(3,2)-g
    gradP1(3,1)*gradP1(1,2)*(basisP1(3)-basisP1(1)) + gradP1(3,1)*basisP1(1)*(gradP1(3,2)-gradP
val(10,:) = [2*basisP1(1)*gradP1(2,1)*gradP1(3,1) + 2*gradP1(1,1)*basisP1(2)*gradP1(3,1) + 2*gr
    2*basisP1(1)*gradP1(2,2)*gradP1(3,2) + 2*gradP1(1,2)*basisP1(2)*gradP1(3,2) + 2*gradP1(1,2)
    basisP1(1)*gradP1(2,1)*gradP1(3,2) + basisP1(1)*gradP1(2,2)*gradP1(3,1) + gradP1(1,1)*basisP
end

%% P1 basis functions
function val = basisP1(x,y)
lambda1 = 1 - x - y;
lambda2 = x;
lambda3 = y;
val = [lambda1,lambda2,lambda3];
end

function val = gradP1(x,y)
val = zeros(size(x,1),3,2);
val(:,1,:) = [-ones(size(x,1)),-ones(size(y,1))];
val(:,2,:) = [ones(size(x,1)),zeros(size(y,1))];
val(:,3,:) = [zeros(size(x,1)),ones(size(y,1))];
end

function [c21,c31] = transformation(c4n,n4e)
n1 = c4n(n4e(:,1),:);
n2 = obj.c4n(n4e(:,2),:);
n3 = obj.c4n(n4e(:,3),:);
area4e = computeArea4e(c4n,n4e);
c = (0.5./area4e)*ones(1,2);
c21 = c.*(n2-n1);
c31 = c.*(n3-n1);
end

function [I,J] = localDoFtoGlobalDoF(dof4e1,dof4e2)

 if nargin < 2
     dof4e2 = dof4e1;
 end

 nrDoF1 = size(dof4e1,2);
 nrDoF2 = size(dof4e2,2);

 dof4e1 = dof4e1';
 I = repmat(dof4e1,nrDoF2,1);
```

```
 I = I(:);

 dof4e2 = dof4e2';
 J = (dof4e2(:)*ones(1,nrDoF1))';
 J = J(:);

end

function val = basisSigma(x,y,basisCoefficients,n4e)
bP3 = basisP3(x,y);
nrElems = size(n4e,1);
bP3 = (bP3(:)*ones(1,nrElems))';
a = basisCoefficients(:,:,1:3:end);
b = basisCoefficients(:,:,2:3:end);
c = basisCoefficients(:,:,3:3:end);
val = zeros(nrElems,3,24);
val(:,1,:) = matMul(a,bP3);
val(:,2,:) = matMul(b,bP3);
val(:,3,:) = matMul(c,bP3);
end

function val = computeEdNumber(e4s,s4e)
edNr4ed = zeros(size(e4s));
ind = (1:size(e4s,1))';
indz = find(e4s(:,2)>0);
edNr4ed(:,1) = mod(find(s4e(e4s(:,1),:)'==(ind*ones(1,3))')-1,3)+1;
edNr4ed(indz,2) = mod(find(s4e(e4s(indz,2),:)'==(ind(indz)*ones(1,3))')-1,3)+1;
end

function val = matMul(A,B)
% For given 3-dimensional matrices A ( dim(A) = [n m k] ) and
% B ( dim(B) = [m l k] ) matrixMultiplication computes the elementwise
% matrix product A(k)*B(k)

% Copyright 2007 David Guenther
%
% This file is part of FFW.
%
% FFW is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% FFW is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.

```

```
if numel(size(A)) < 3
    A = permute(A,[2 3 1]);
else
    A = permute(A,[2 3 1]);
end

if numel(size(B)) < 3
    B = permute(B,[2 3 1]);
else
    B = permute(B,[2 3 1]);
end

dimMatrixA = size(A);
dimMatrixB = size(B);

if ( length(dimMatrixA) > 2 && dimMatrixA(3) ~= dimMatrixB(3) )
    error('The third dimension must be equal for both matrices.')
end

if ( length(dimMatrixA) < 3 && length(dimMatrixA) < 3 )
    val = A * B;
elseif ( dimMatrixA(1) == 1 && dimMatrixA(2) == 1 )
        A1 = A(:)';
        A = zeros(dimMatrixB(1)*dimMatrixB(2),length(A1));
        for k = 1:dimMatrixB(1)*dimMatrixB(2)
          A(k,:) = A1;
        end
        % A = repmat(A,dimMatrixB(1)*dimMatrixB(2),1);
        A = reshape(A,[dimMatrixB(1),dimMatrixB(2),dimMatrixB(3)]);
        val = A.*B;
% val = zeros(dimMatrixB);
% parfor k = 1:dimMatrixB(3)
% val(:,:,k) = A(1,1,k).*B(:,:,k);
% end
% return;
elseif ( dimMatrixB(1) == 1 && dimMatrixB(2) == 1 )
    B1 = B(:)';
    B = zeros(dimMatrixA(1)*dimMatrixA(2),length(B1));
    for k = 1:dimMatrixA(1)*dimMatrixA(2)
      B(k,:) = B1;
    end
    % B = repmat(B,dimMatrixA(1)*dimMatrixA(2),1);
    B = reshape(B,[dimMatrixA(1),dimMatrixA(2),dimMatrixA(3)]);
    val = B.*A;
% return;
% val = zeros(dimMatrixA);
% parfor k = 1:dimMatrixA(3)
% val(:,:,k) = B(1,1,k).*A(:,:,k);
% end
else
    if dimMatrixA(2) ~= dimMatrixB(1)
        error('The number of columns of matrix A must be equal with the number of rows of matrix
```

```
    end

    permA = permute(A,[2 1 3]);
    for k = 1:dimMatrixB(2)
        repA((k-1)*dimMatrixA(2)+1:k*dimMatrixA(2),:,:) = permA;
    end
    % repA = repmat(permA,[dimMatrixB(2) 1 1]);
    linA = repA(:);

    for k = 1:dimMatrixA(1)
        repB(:,(k-1)*dimMatrixB(2)+1:k*dimMatrixB(2),:) = B;
    end
    % repB = repmat(B,[1 dimMatrixA(1) 1]);
    linB = repB(:);


    val = linA.*linB;
    val = reshape(val,dimMatrixA(2),[]);
    val = sum(val,1)';
    val = reshape(val,dimMatrixB(2),dimMatrixA(1),[]);
    val = permute(val,[2 1 3]);
% val = zeros(dimMatrixA(1),dimMatrixB(2),dimMatrixB(3));
% parfor k = 1:dimMatrixB(3)
% val(:,:,k) = A(:,:,k)*B(:,:,k);
% end
end

val = permute(val,[3 1 2]);
end

function result = rowaddr(A,I,J,S)
% rowaddr.m pointwise reads entries [I,J] from A if S is omitted, i.e.,
% result = A( I(j,k), J(j,k)).
% If S is submitted, then entries [I,J] from A are pointwise set to S and
% the new A is returned as a result.

% Copyright 2007 Jan Reininghaus, David Guenther, Andreas Byfut
%
% This file is part of FFW.
%
% FFW is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% FFW is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
```

```
557
    if nargin == 3
        % initialise result
        result = zeros(size(I));
        % if size of A is small enough we can use linear adressing
562     if(numel(A) < 2^31)
            for curCol = 1:size(I,2)
                linIndex = sub2ind(size(A),I(:,curCol),J(:,curCol));
                result(:,curCol) = full( A(linIndex) );
            end
567     else
            for j = 1:size(I,2)
                dummyI = I(:,j);
                dummyJ = J(:,j);
                dummyres = zeros(size(I,1),1);
572             for k = 1:size(I,1)
                    dummyres(k) = A(dummyI(k),dummyJ(k));
                end
                result(:,j) = dummyres;
            end
577     end
    elseif nargin == 4
        % initialise result
        result = A;
        % if size of A is small enough we can use linear adressing
582     if(numel(A) < 2^31)
            for curCol = 1:size(I,2)
                linIndex = sub2ind(size(A),I(:,curCol),J(:,curCol));
                result(linIndex) = S(:,curCol);
            end
587     else
            for j = 1:size(I,2)
                dummyI = I(:,j);
                dummyJ = J(:,j);
                dummyS = S(:,j);
592             for k = 1:size(I,1)
                    result(dummyI(k),dummyJ(k)) = dummyS(k);
                end
            end
        end
597 else
        if nargin < 3
            error('Not enough parameters submitted,');
        end
    end
602 end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Copyright 2009-2015
    % Numerical Analysis Group
607 % Prof. Dr. Carsten Carstensen
    % Humboldt-University
```

```
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.79: solve/solveCRP0Stokes.m

```
function [u,p,A,b,nrDofs,gradU4e]=solveCRP0Stokes(f,u4sDb,g,c4n,n4e,n4sDb,n4sNb)
%% solveCRP0Stokes - solve the Stokes problem using the CR element.
% Solves the Stokes problem on the domain given by c4n and n4e for
% right-hand side f, Dirichlet boundary condition u4sDb on n4sDb and
% Neumann boundary condition g on n4sNb.
%
% Input: f right-hand side of the problem definition
% u4Db Dirichlet boundary condition
% g Neumann boundary condition
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: u basis coefficients of the numerical solution of the
% velocity field w.r.t. the Crouzeix-Raviart basis
% p basis coefficients of the numerical solution of the
% pressure w.r.t. the P0 basis
% A the matrix A of the linear system created
% b the right side of the linear system created
% nrDofs number of degrees of freedom
% gradU4e piecewise gradient of the discrete solution u

    %% Initialization
    n4s=computeN4s(n4e);
    mid4s=computeMid4s(c4n,n4s);
    s4e=computeS4e(n4e);
    s4n=computeS4n(n4e);
    nrSides=size(n4s,1);
    nrElems=size(n4e,1);
    area4e=computeArea4e(c4n,n4e);
```

```
    length4s=computeLength4s(c4n,n4s);

    %% Assembling local A,B
    A4e=zeros(3,3,nrElems);
    B4e=zeros(1,6,nrElems);
    grads4e=zeros(3,2,nrElems);
    for j=1:nrElems
        grads=[c4n(n4e(j,:),:)'; 1 1 1]\[-2 0; 0 -2; 0 0];
        A4e(:,:,j)=area4e(j)*(grads*grads');
        B4e(:,:,j)=-area4e(j)*grads(:)';
        grads=grads([3 1 2],:);
        grads4e(:,:,j) = grads;
    end

    %% Assembling global A,B
    dofs_u=s4e(:,[2 3 1])';
    I=repmat(dofs_u(:),1,size(dofs_u,1))';
    J=repmat(dofs_u',1,size(dofs_u,1))';
    A1D=sparse(I(:),J(:),A4e(:));
    A=[A1D sparse(nrSides,nrSides)
       sparse(nrSides,nrSides) A1D];

    dofs_u=[s4e(:,[2 3 1]) nrSides+s4e(:,[2 3 1])];
    dofs_p=1:nrElems;
    I=repmat(dofs_p(:),1,size(dofs_u,2))';
    J=repmat(dofs_u,1,size(dofs_p,1))';
    B=sparse(I(:),J(:),B4e(:),nrElems,2*nrSides);

    %% RHS vector b for zero boundary conditions
    mid4e=computeMid4e(c4n,n4e);
    f4e=f(mid4e).*[area4e area4e]/3;
    b4e=[f4e(:,1) f4e(:,1) f4e(:,1) f4e(:,2) f4e(:,2) f4e(:,2)];
    b=accumarray(dofs_u(:),b4e(:));
    b=[b; zeros(nrElems+1,1)];

    %% Modify b for nonhomogenous boundary conditions
    % Dirichlet boundary conditions
    DbSides=zeros(1,size(n4sDb,1));
    for j=1:size(n4sDb,1);
        DbSides(j)=s4n(n4sDb(j,1),n4sDb(j,2));
    end
    l4DbS=computeLength4s(c4n,n4sDb);
    freeSides=setdiff(1:size(n4s,1),DbSides);

    mean4DbSides=integrate(c4n,n4sDb,@(x,y,z)(u4sDb(y)),10)./[l4DbS l4DbS];
                                    % calculate integral means of u_D
                                    % along DirichletSides
    x=zeros(2*nrSides+nrElems+1,1);
    x([DbSides nrSides+DbSides])=[mean4DbSides(:,1)' mean4DbSides(:,2)'];
    b=b-[A B' sparse(2*nrSides,1)]'*x(1:2*nrSides);

    % Neumann boundary conditions
```

```
    for j=1:size(n4sNb,1)
        side=s4n(n4sNb(j,1),n4sNb(j,2));
        b([side nrSides+side])=b([side nrSides+side]) +...
                                    length4s(side)*g(mid4s(side,:))';
    end

    %% Solve the linear system
    dofs=[freeSides nrSides+freeSides 2*nrSides+(1:nrElems) ...
                                                    2*nrSides+nrElems+1];
    M=[A B' sparse(2*nrSides,1);B sparse(nrElems,nrElems) area4e;...
                                        sparse(1,2*nrSides) area4e' 0];
    x(dofs)=M(dofs,dofs)\b(dofs);
    nrDofs=length(dofs);
    u1=x(1:nrSides);
    u2=x(nrSides+1:2*nrSides);
    u=[u1 u2];
    p=x(2*nrSides+1:2*nrSides+nrElems);

    %% (optional) Computation of nonconforming derivative
    if nargout > 5
        gradU4e = zeros(2,2,nrElems);
        for j = 1:nrElems
            gradU4e(1,:,j) = u1(s4e(j,:))' * grads4e(:,:,j);
            gradU4e(2,:,j) = u2(s4e(j,:))' * grads4e(:,:,j);
        end
    end

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
```

```
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.80: solve/solveCRPoisson.m

```
function [x,nrDof,A,b] = solveCRPoisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)
%% solveCR - solve the Possion problem using the CR element.
% Solves the Poisson problem for given right-hand side f and
% Neumann boundary condition g on the domain given by c4n and n4e.
%
% Input: f right-hand side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: x basis coefficients of the numerical solution w.r.t.
% the Crouzeix-Raviart basis.
% nrDof number of degrees of freedom
% A the matrix A of the linear system created
% b the right side of the linear system created

    %% Initialisation
    nrElems = size(n4e,1);
    s4e = computeS4e(n4e);
    nrSides = max(max(s4e));
    area4e = computeArea4e(c4n,n4e);
```

```
    mid4e = computeMid4e(c4n,n4e);
    s4n = computeS4n(n4e);
    % Dirichlet boundary sides
    DbSides = zeros(1,size(n4sDb,1));
    for i = 1:size(n4sDb,1)
        DbSides(i) = s4n(n4sDb(i,1),n4sDb(i,2));
    end
    % Neumann boundary sides
    NbSides = zeros(1,size(n4sNb,1));
    for i = 1:size(n4sNb,1)
        NbSides(i) = s4n(n4sNb(i,1),n4sNb(i,2));
    end
    % degrees of freedom: one per non-Dirichlet side
    dof = setdiff(1:nrSides,DbSides);
    nrDof = length(dof);

    Alocal = zeros(3,3,nrElems);
    b = zeros(nrSides,1);

    %% Create the stiffness matrix A and right-hand side b
    for elem = 1 : nrElems
        nodes = n4e(elem,:); % nodes of this element
        sides = s4e(elem,:); % sides of this element
        coords = c4n(nodes,:); % coordinates for the nodes
        area = area4e(elem); % area of this element
        grads = [coords';1 1 1]\[-2 0; 0 -2; 0 0]; % gradients for CR basis
        grads = grads([3 1 2],:); % reorder to fit DoF numbering
        Alocal(:,:,elem) = area * grads * grads'; % local stiffness matrix
        mid = mid4e(elem,:); % midpoint of this element
        b(sides) = b(sides) + area*f(mid)*ones(3,1)/3; % right-hand side
    end

    % assembly of the global stiffness matrix A
    s4eT = s4e';
    I = [s4eT;s4eT;s4eT];
    J = [s4eT(:),s4eT(:),s4eT(:)]';
    A = sparse(I(:),J(:),Alocal(:));

    %% Neumann boundary conditions
    length4NbSides = computeLength4s(c4n,n4sNb);
    mid4NbSides = computeMid4s(c4n,n4sNb);
    b(NbSides) = b(NbSides) + length4NbSides .* g(mid4NbSides);
    %% Dirichlet boundary conditions
    x = zeros(nrSides,1);
    x(DbSides) = u4Db(computeMid4s(c4n, n4sDb));
    b = b - A * x;
    %% solve the algebraic equation
    x(dof) = A(dof,dof) \ b(dof);

end

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

```
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.81: solve/solveCRPoissonNEU.m

```
function [x,nrDof,A,b] = solveCRPoissonNEU(f,g,u_D,c4n,n4e,Db,Nb)


edge_nr = computeS4N(n4e);
L_tmp = max(max(edge_nr)); L = L_tmp(1);
A = sparse(L,L); b = zeros(L,1); x = zeros(L,1);

DiriNodes = diag(edge_nr(Db(:,1),Db(:,2)));
freeNodes = setdiff([1:L],DiriNodes);
shift1 = [2,3,1]; shift2 = [3,1,2];

for j = 1 : size(n4e,1)
    grads_T = [1,1,1;c4n(n4e(j,:),:)'] \ [0,0;eye(2)];
    nc_grads_T = [-1,1,1;1,-1,1;1,1,-1] * grads_T;
    area_T = det([1,1,1;c4n(n4e(j,:),:)'])/2;
    mp_T = sum(c4n(n4e(j,:),:))/3;
    for m = 1 : 3
        I = edge_nr(n4e(j,shift1(m)),n4e(j,shift2(m)));
        b(I) = b(I) + area_T * f(mp_T) / 3;
        for n = 1 : 3
            J = edge_nr(n4e(j,shift1(n)),n4e(j,shift2(n)));
            A(I,J) = A(I,J) + area_T * nc_grads_T(m,:) * nc_grads_T(n,:)';
        end
    end
end

for j = 1 : size(Nb,1)
```

```
    length_E = norm(c4n(Nb(j,1),:) - c4n(Nb(j,2),:));
    mp_E = (c4n(Nb(j,1),:) + c4n(Nb(j,2),:))/2;
    I = edge_nr(Nb(j,1),Nb(j,2));
    b(I) = b(I) + length_E * g(mp_E);
end

x(DiriNodes) = u_D((c4n(Db(:,1),:) + c4n(Db(:,2),:))/2);
b = b - A * x;
x(freeNodes) = A(freeNodes,freeNodes) \ b(freeNodes);
% show(c4n,n4e,edge_nr,x);

nrDof = numel(freeNodes);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function val = g(x);
val = 1;
function val = u_D(x);
val = 0;
function val = f(x);
val = exp(-1/x(1));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% function show(c4n,n4e,edge_nr,x);
% clf; hold on; view(30,30)
% shift1 = [2,3,1]; shift2 = [3,1,2];
% for j = 1 : size(n4e,1)
% for m = 1 : 3
% I = edge_nr(n4e(j,shift1(m)),n4e(j,shift2(m)));
% z(m) = x(I);
% end
% u = [-1,1,1;1,-1,1;1,1,-1] * z';
% trisurf([1,2,3],c4n(n4e(j,:),1),c4n(n4e(j,:),2),u,1);
% end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% function edge_nr = edge_numbers(n4e,c4n);
% shift1 = [2,3,1];
% edge_nr = sparse(size(c4n,1),size(c4n,1));
% nr_edges = 0;
% for j = 1 : size(n4e,1)
% for k = 1 : 3
% if ~edge_nr(n4e(j,k),n4e(j,shift1(k)))
% nr_edges = nr_edges + 1;
% edge_nr(n4e(j,shift1(k)),n4e(j,k)) = nr_edges;
% end
% end
% end
% edge_nr = edge_nr + edge_nr';
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.82: solve/solveP1P1Elasticity.m

```
function [x,nrDof,A,b] = solveP1P1Elasticity(f,g,u4Db,c4n,n4e,n4sDb,n4sNb,mu,lambda)
    %% Initialisation
    A = sparse(2*size(c4n,1),2*size(c4n,1));
    b = zeros(2*size(c4n,1),1);
    nrNodes = size(c4n,1);
    DirichletNodes = unique(n4sDb);
    dof = setdiff(1:nrNodes,DirichletNodes); % free nodes to be approximated
    nrDof = length(dof);

    % Assembly
    for j = 1:size(n4e,1)
        I = 2*n4e(j,[1,1,2,2,3,3]) -[1,0,1,0,1,0];
        A(I,I) = A(I,I) +stima3(c4n(n4e(j,:),:),lambda,mu);
    end

    % Volume forces
    for j = 1:size(n4e,1)
        I = 2*n4e(j,[1,1,2,2,3,3]) -[1,0,1,0,1,0];
        fs = f(sum(c4n(n4e(j,:),:))/3)';
        b(I) = b(I) +det([1,1,1;c4n(n4e(j,:),:)'])*[fs;fs;fs]/6;
    end

    % Neumann conditions
    if ~isempty(n4sNb)
        n = (c4n(n4sNb(:,2),:) -c4n(n4sNb(:,1),:))*[0,-1;1,0];
        for j = 1:size(n4sNb,1);
```

```
            I = 2*n4sNb(j,[1,1,2,2]) -[1,0,1,0];
            gm = g(sum(c4n(n4sNb(j,:),:))/2, n(j,:)/norm(n(j,:)))';
            b(I) = b(I) +norm(n(j,:))*[gm;gm]/2;
        end
    end

    % Dirichlet conditions (Note: For this particular example, the Lame
    % constants LAMBDA and MU are also needed for the Dirichlet boundary
    % condition.)
    [W,M] = u4Db(c4n(DirichletNodes,:),lambda,mu);
    B = sparse(size(W,1),2*size(c4n,1));
    for k = 0:1
        for l = 0:1
            B(1+l:2:size(M,1),2*DirichletNodes-1+k) = diag(M(1+l:2:size(M,1),1+k));
        end
    end
    mask = find(sum(abs(B)'));
    A = [A, B(mask,:)'; B(mask,:), sparse(length(mask),length(mask))];
    b = [b;W(mask,:)];

    % Calculating the solution
    x = A \ b;
    x = x(1:2*size(c4n,1)); %Remove Lagrange multipliers
end


%% Additional functions
function stima3=stima3(vertices,lambda,mu)
    %STIMA3 Computes element stiffness matrix for triangles.
    % M = STIMA3(X,LAMBDA,MU) computes element stiffness matrix for
    % triangles. The coordinates of the vertices are stored in X. LAMBDA
    % and MU are the Lame constants.
    %
    % This routine should not be modified.
    %
    %
    % See also FEM_LAME2D and STIMA4.

    % J. Alberty, C. Carstensen and S. A. Funken 07-03-00
    % File <stima3.m> in $(HOME)/acfk/fem_lame2d/cooks/ and
    % $(HOME)/acfk/fem_lame2d/lshape_p1/ and
    % $(HOME)/acfk/fem_lame2d/lshape_q1/ and
    % $(HOME)/acfk/fem_lame2d/hole/

    PhiGrad = [1,1,1;vertices']\[zeros(1,2);eye(2)];
    R = zeros(3,6);
    R([1,3],[1,3,5]) = PhiGrad';
    R([3,2],[2,4,6]) = PhiGrad';
    C = mu*[2,0,0;0,2,0;0,0,1] +lambda*[1,1,0;1,1,0;0,0,0];
    stima3 = det([1,1,1;vertices'])/2*R'*C*R;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.83: solve/solveP1Poisson.m

```
function [x,nrDof,A,b] = solveP1Poisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)
% solveP1Poisson.m
% solves the Poisson problem for given righthand side f and
% Neumann boundary condition g on the domain given by c4n and n4e.
%
% Input: f right side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: x solution u for each node corresponding to c4n
% nrDof number of degrees of freedom
% A the matrix A of the linear system created (optional)
% b the right side of the linear system created (optional)

    %% Initialisation
    nrNodes = size(c4n,1); % number of nodes
    nrElems = size(n4e,1); % number of elements
    DbNodes = unique(n4sDb); % Dirichlet boundary nodes
    dof = setdiff(1:nrNodes,DbNodes); % free nodes to be approximated
    nrDof = length(dof);
    Alocal = zeros(3,3,nrElems); % local stiffness matrices
    b = zeros(nrNodes,1); % vector for right-hand side
```

```
    %% Create the stiffness matrix A and right-hand side b
    area4e = computeArea4e(c4n,n4e);
    mid4e = computeMid4e(c4n,n4e);
    % calculate for each element the gradients of the three nodal
    % basisfunctions, the right-hand side and the local stiffness matrix A
    for elem = 1 : nrElems
        nodes = n4e(elem,:); % nodes of the triangle
        coords = c4n(nodes,:); % coordinates of the three nodes
        area = area4e(elem); % area of the current element
        mid = mid4e(elem,:); % midpoint of the triangle
        grads = [1,1,1;coords'] \ [0,0;eye(2)];
        b(nodes) = b(nodes)+(1/3) * area * f(mid)*[1;1;1];
        Alocal(:,:,elem) = area * grads * grads';
    end

    % assembly of the global stiffness matrix A
    n4eT = n4e';
    I = [n4eT;n4eT;n4eT];
    J = [n4eT(:),n4eT(:),n4eT(:)]';
    A = sparse(I(:),J(:),Alocal(:));

    %% Neumann boundary conditions
    % involving Neumann boundary for all Neumann edges
    nrNbSides = size(n4sNb,1);
    length4NbSides = computeLength4s(c4n,n4sNb);
    mid4NbSides = computeMid4s(c4n,n4sNb);
    for NbSide = 1 : nrNbSides
        nodes = n4sNb(NbSide,:); % nodes of the edge
        len = length4NbSides(NbSide); % length of the edge
        mid = mid4NbSides(NbSide,:); % midpoint of the edge
        b(nodes) = b(nodes) + (1/2) * len * g(mid)*[1;1];
    end

    %% Dirichlet boundary conditions
    x = zeros(nrNodes,1); % get the Dirichlet values
    DbCoords = c4n(DbNodes,:); % coordinates of Dirichlet nodes
    x(DbNodes) = u4Db(DbCoords);
    b = b - A * x; % substract inhomogenous boundary

    %% solve the algebraic equation
    x(dof) = A(dof,dof) \ b(dof);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
```

```
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 3.84: solve/solveRT0Poisson.m

```
function [p,u,nrDof] = solveRT0Poisson(f,g,u4Db,c4n,n4e,n4sDb,n4sNb)
% solveRT0Poisson.m
% solves the Poisson problem for given right-hand side f and
% Neumann boundary condition g on the domain given by c4n and n4e.
% The realised implementation is based on the Lagrange Multiplier technique
%
%
% Input: f right side of the problem definition
% g Neumann boundary condition
% u4Db Dirichlet boundary condition
% c4n coordinates for the nodes of the mesh
% n4e nodes for the elements of the mesh
% n4sDb the nodes of the sides in the Dirichlet boundary
% n4sNb the nodes of the sides in the Neumann boundary
%
% Output: x solution x = (u,p) from RT0 and P0;
% u includes an vector for each element,
% while p includes an value for each element
% nrDof number of degrees of freedom
% A the matrix A of the linear system created (optional)
% b the right side of the linear system created
% (optional)

    %% Initialisation
    n4s = computeN4s(n4e); % Nodes for sides
    s4n = computeS4n(n4e); % Sides for nodes
    s4e = computeS4e(n4e); % Sides for elements
    e4s = computeE4s(n4e); % Elements for sides
    area4e = computeArea4e(c4n,n4e); % Area for elements
    mid4e = computeMid4e(c4n,n4e); % Midpoints of elements
    mid4s = computeMid4s(c4n,n4s); % Midpoints of sides
    length4s= computeLength4s(c4n,n4s); % Length of Sides
    normal4s = computeNormal4s(c4n,n4s); % Normals for sides

    nrElems = size(n4e,1); % Number of elements
```

```
36      nrISides = length(find(e4s(:,2)~=0)); % Number of inner Sides
        nrNbSides = size(n4sNb,1); % Number of Neumann boundary sides
        nrDbSides = size(n4sDb,1); % Number of Dirichlet boundary sides

        n4iSides = n4s(e4s(:,2)~=0,:); % Nodes for interiour sides
41
        % Blocks of the global stiffnes Matrices
        B = sparse( 3*nrElems, 3*nrElems);
        C = sparse( 3*nrElems, nrElems);
        D = sparse( 3*nrElems, nrISides);
46      F = sparse( 3*nrElems, nrNbSides);

        b = zeros(4*nrElems+nrISides+nrNbSides,1); % right-hand side (RHS)

        % Assembling the blocks B, C and the RHS
51      for curElem = 1 : nrElems
            % Summing up the length (norm) of all sides of curElem
            s = sum(length4s(s4e(curElem,:)).^2);
            % assembling the blocks of global stiffness matrices B and C
            B( 3*curElem-[2,1,0],3*curElem-[2,1,0] ) = area4e(curElem) * diag([1,1,s/36]);
56          C( 3*curElem-[2,1,0],curElem ) = [0;0;2*area4e(curElem)];
            % assembling the RHS b
            b(3*nrElems+curElem) = -area4e(curElem) * f(mid4e(curElem,:));
        end

61      %% Assembling the block D (Condition for interior sides)
        for curISide = 1 : nrISides
            side = s4n(n4iSides(curISide,1),n4iSides(curISide,2));
            curNormal = normal4s(side,:)';
            h1 = (c4n(n4iSides(curISide,1),:)-mid4e(e4s(side,1),:)) * curNormal;
66          h2 = (c4n(n4iSides(curISide,1),:)-mid4e(e4s(side,2),:)) * curNormal;
            % assembling the blocks of global stiffness matrix D
            D( [3*e4s(side,1)-[2,1,0],3*e4s(side,2)-[2,1,0]],curISide ) ...
               = -length4s(side)*[curNormal;h1;-curNormal;-h2];
        end
71
        %% Dirichlet conditions; Assembly of b
        for curDbSide = 1 : nrDbSides
            side = s4n(n4sDb(curDbSide,1),n4sDb(curDbSide,2));
            curNormal = normal4s(side,:)';
76          curElement = e4s(side,1);
            h = (c4n(n4sDb(curDbSide,1),:)-mid4e(curElement,:)) * curNormal;
            b(3*curElement-[2,1,0]) = b(3*curElement-[2,1,0]) ...
                + u4Db(mid4s(side,:)) *length4s(side)*[curNormal;h];
        end
81
        %% Neumann conditions; Assembling block F
        for curNbSide = 1 : nrNbSides
            side = s4n(n4sNb(curNbSide,1),n4sNb(curNbSide,2));
            curNormal = normal4s(side,:)';
86          curElem = e4s(side,1);
            h = (c4n(n4sNb(curNbSide,1),:)-mid4e(curElem,:)) * curNormal;
```

```
        F(3*curElem-[2,1,0],curNbSide) = length4s(side)*[curNormal;h];
        b(4*nrElems+nrISides+curNbSide) = length4s(side)*g(mid4s(side,:));
    end

    %% Assembling the global stiffness matrix, including zero block matrices
    O1 = sparse(size(C,2),size(C,2)+size(D,2)+size(F,2));
    O2 = sparse(size(D,2),size(C,2)+size(D,2)+size(F,2));
    O3 = sparse(size(F,2),size(C,2)+size(D,2)+size(F,2));
    A = [B , C, D, F; ...
         C', O1 ; ...
         D', O2 ; ...
         F', O3 ];

    nrDof = nrISides + nrNbSides;

    %% Solve the linear System of equations
    x = A \ b;

    p = reshape(x(1:3*nrElems),3,nrElems)';
    u = x(3*nrElems+1:4*nrElems);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright 2009-2015
% Numerical Analysis Group
% Prof. Dr. Carsten Carstensen
% Humboldt-University
% Departement of Mathematics
% 10099 Berlin
% Germany
%
% This file is part of AFEM.
%
% AFEM is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 3 of the License, or
% (at your option) any later version.
%
% AFEM is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program. If not, see <http://www.gnu.org/licenses/>.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Bibliography

[1] C. Aguilera. quiver2.m. extern/quiver2/, Matlab Central (e.g., http://www.mathworks.com/matlabcentral/fileexchange/24600). 50

[2] M. Ainsworth and J. T. Oden. *A posteriori error estimation in finite element analysis*. Wiley-Interscience [John Wiley & Sons], New York, 2000. 22

[3] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numer. Algorithms*, 20(2–3):117–137, 1999.

[4] C. Bahriawati and C. Carstensen. Three Matlab implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Computational Methods in Applied Mathematics*, 5:333–361, 2005. 14, 17, 18

[5] W. Bangerth and R. Rannacher. *Adaptive finite element methods for differential equations*. Lectures in Mathematics ETH Zürich, Basel, 2003. 22

[6] P. Binev, W. Dahmen, and R. D. Vore. Adaptive finite element methods with convergence rates. *Numer. Math.*, 97:219–268, 2004. 31

[7] D. Braess. *Finite Elements*. Cambridge University Press, 2007. 7

[8] S. C. Brenner and C. Carstensen. *Encyclopedia of Computational Mechanics*, chapter 4, Finite Element Methods. John Wiley and Sons, 2004. 23, 31

[9] C. Carstensen. An adaptive mesh-refining algorithm allowing for an H^1 stable L^2 projection onto courant finite element spaces. *Constructive Approximation*, 20:549–564, 2004. 31

[10] C. Carstensen, S. Bartels, and S. Jansche. A posteriori error estimates for nonconforming finite element methods. *Numer. Math.*, 92:233–256, 2002. 24

[11] A. Stroud. *Approximate Calculations of Multiple Integrals*. Prentice-Hall, Englewood Cliffs, 1971. 36

[12] R. Verfürth. *A review of a posteriori error estimation and adaptive mesh-refinement techniques*. Wiley-Teubner Series Advances in Numerical Mathematics, 1996. 22