Unterschrift des Betreuers

Diplomarbeit

A Space-Time Adaptive Algorithm for Linear Parabolic Problems

Ausgeführt am

Institut für
Analysis und Scientific Computing
der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.-Prof. Dipl.-Math.Dr.techn. Dirk Praetorius

durch

Philipp Wissgott
A-1160 Wien

Datum

Unterschrift

Kurzfassung

In Wissenschaft und Technik gewinnt die numerische Simulation komplexer physikalischer Vorgänge zunehmend an Bedeutung. Eine wesentliche Rolle spielt dabei das mathematische Maß der Genauigkeit einer numerischen Lösung, um die Zuverlässigkeit der so gewonnenen Approximation zu beurteilen. Neben der Zuverlässigkeit ist auch die Effizienz einer Simulationssoftware von zentraler Bedeutung. So will man beispielsweise unter gegebenen Restriktionen an die Speicher- oder Rechenkapazität eine möglichst gute numerische Lösung erhalten. Eine mathematische Fundierung dieser beiden Aspekte einer Simulation beruht üblicherweise auf sogenannten a posteriori Fehlerabschätzungen: Basierend auf der mathematisch-physikalischen Problemstellung und den gegebenen (und somit bekannten) Daten versucht man eine numerische Lösung u_h zu berechnen, sodass man den Fehler $u - u_h$ in einem geeigneten Maß abschätzen kann, ohne die exakte Lösung u kennen zu müssen.
In der vorliegenden Arbeit werden modellhaft lineare parabolische Differentialgleichungen mit zeit- und ortsabhängigen Koeffizienten betrachtet. Solche Probleme treten z.B. bei der Berechnung von Temperaturverteilungen durch Wärmeleitung auf. Auch die Ausbreitung von Verunreinigungen in Grundwasser lässt sich analog modellieren.
In dieser Arbeit wird zunächst der mathematische Rahmen für die Analysis elliptischer und parabolischer Differentialgleichungen wiederholt. Die Diskretisierung der kontinuierlichen parabolischen Gleichung erfolgt mit Hilfe der Linienmethode. Wir verwenden das implizite Euler-Verfahren in der Zeit und eine konforme P^1-Finite Elemente Methode im Ort.
Ausgehend von einer Arbeit von Verfürth [16] für parabolische Gleichungen mit konstanten Koeffizienten leiten wir eine a posteriori Fehlerabschätzung für Probleme mit zeit- und ortsabhängigen Koeffizienten her, die den Gesamtfehler $|||u(T) - u_h(T)|||$ zu einem Zeitpunkt $T > 0$ in der Energienorm des Problems zuverlässig und effizient schätzt. Es ist wohlbekannt, dass die duale Norm des Residuums äquivalent zum Fehler in der Energienorm ist. Verfürths Idee ist es, das Residuum als Summe $R = R_h + R_\tau + R_D$ von Orts-, Zeit- und Datenresiduum darzustellen. Dabei wird das Ortsresiduum R_h gerade so definiert, dass es auf jedem Zeitintervall die übliche Galerkin-Orthogonalität erfüllt. Dies erlaubt die Anwendung analytischer Standardtechniken aus dem Kontext elliptischer Differentialgleichungen, um die duale Norm von R_h durch a posteriori berechenbare Terme nach oben und unten abzuschätzen. Die Abschätzung des Zeitresiduums R_τ ist technisch, aber elementar.
Eine wesentliche Aufgabe der Arbeit war die Umsetzung der hergeleiteten Fehlerabschätzung in Form eines adaptiven Algorithmus zur numerischen Lösung linearer parabolischer Gleichungen. Dabei werden im Lösungsprozess sowohl die Zeitschrittweite als auch die lokale Netzweite der Ortsdiskretisierung selbstständig und unabhängig voneinander an eine vorgegebene Toleranz angepasst. Dabei verwenden wir die Heuristik, dass die räumlichen und zeitlichen Beiträge des Residuums unabhängig voneinander sind. Die entwickelte adaptive Strategie erlaubt insbesondere die Verfeinerung und Vergröberung des Netzes in aufeinanderfolgenden Zeitschritten.
Die Implementierung des adaptiven Algorithmus erfolgt in MATLAB. Bei der Realisierung wird darauf geachtet, MATLABs effiziente Vektorrechnung möglichst weitgehend auszunützen. Speziell werden alle auftretenden Matrizen vektorisiert und insbesondere ohne Verwendung von Schleifen assembliert. Besonderes Augenmerk liegt auf einer möglichst allgemeinen Form des adaptiven Algorithmus. Dadurch ergibt sich ein bestimmtes Maß an Ineffizienz für Probleme mit konstanten Koeffizienten, aufgrund überflüssiger Funktionsauswertungen.
Abschließend testen wir den vorgestellten Algorithmus in numerischen Experimenten. Dabei zeigt sich, dass die adaptive Strategie auch in praktischen Beispielen einer uniformen Strategie weit überlegen ist. Andererseits führt die verwendete Netzverwaltungsstrategie bei manchen Problemen zu einer gewissen Ineffizienz.

Contents

Introduction

For the last 50 years, the finite element method has been used to solve partial differential equations in applied mathematics, but also in civil engineering and fluid dynamics. Since the early work of Ciarlet [8], Babuška [3], and Clément [9], with an increasing number of applications, this technique offers a fast and robust way to solve elliptic and parabolic problems numerically. Based on variational principles which date back to Ritz [14], the finite element method provides a simple and intuitive setting with direct links to energy methods used by physicists. Though solving elliptic problems with finite elements has been generally understood in the course of the previous decades, parabolic problems are still subject of current research, e.g. Eriksson [10] or Johnson [6].

Scope

Most of the theoretical research in the field of linear parabolic equations is restricted to model problems with trivial or constant coefficients. The goal of this work is to go beyond that setting and admit time and space dependent coefficients in order to cover more general classes of linear parabolic equations. Based on work of Verfürth [16], we derive a reliable residual-based a posteriori error estimator, which is global in time and space. Moreover, we analyze efficiency of this estimator which is local in time and global in space, if certain data functions are independent of the space variable.
We develop efficient techniques to compute this a posteriori error estimator, and we provide a space-time adaptive grid strategy to solve a wide range of linear parabolic equations. The implementation is done in MATLAB, which we regard as an appropriate tool for the development and the testing of modern numerical algorithms.

Outline of the Work

Here, we restrict ourselves to two dimensional Lipschitz-domains $\Omega \subset \mathbb{R}^2$, and use triangles as elements in the spatial discretization $\mathcal{T}$ of this domain.

In Chapter 1 the triangulations $\mathcal{T}$ we are working with are specified. In contrast to the standard regularity assumptions, these triangulations may include triangles with a hanging node. We also describe the necessary notations for the grid management. To improve the quality of the discrete solution, it is usually necessary to refine a given triangulation $\mathcal{T}$. On the other hand, it may be appropriate to coarsen the mesh for problems, where the solution becomes smoother with respect to time. We discuss in great detail our strategy for grid-refinement and grid-coarsening, and we show that both do not crucially affect the shape regularity of $\mathcal{T}$, i.e. the ratios $h_K^2/|K|$ of the triangles $K \in \mathcal{T}$ can be uniformly controlled. This attribute is decisive, since most of the estimates of the a priori and the a posteriori error analysis depend on the

so-called shape regularity constant $c_s(\mathcal{T}) := \max_{K\in\mathcal{T}} h_K^2/|K|$.

At the end of this chapter, we discuss the implemented data structures as well as the MATLAB code of the functions for refinement and coarsening.

The mathematical background of the finite element method, including the definition of the function spaces, is recalled in Chapter 2. We start with the considerations of the elliptic model problem

$$\begin{aligned} -\operatorname{div}(D\nabla u) + \vec{c}\cdot\nabla u + ru &= f \quad \text{in } \Omega,\\ u &= 0 \quad \text{on } \Gamma_D,\\ \vec{n}\cdot D\nabla u &= g \quad \text{on } \Gamma_N, \end{aligned}$$

with in general unknown solution u and given data D, $\vec{c}$, r, f functions of x in Ω and with g a function of x on Γ_N. This elliptic problem corresponds to the steady state of the parabolic problem introduced below. By integrating the differential equation on Ω with a test function $v \in H^1_D(\Omega)$, we find the weak form

$$-\int_\Omega \operatorname{div}(D\nabla u)\, v\, dx + \int_\Omega \vec{c}\cdot\nabla u\, v\, dx + \int_\Omega r\, u\, v\, dx = \int_\Omega f\, v\, dx.$$

Here, $H^1_D(\Omega)$ denotes the subspace of the Sobolev space $H^1(\Omega) = W^{1,2}(\Omega)$, with functions that vanish on the Dirichlet boundary Γ_D. We recapitulate existence and uniqueness results for the weak solution $u \in H^1_D(\Omega)$. Furthermore, we define the finite element space $\mathcal{S}^1_D$, which consists of $\mathcal{T}$-piecewise affine and globally continuous functions, which are trivial on the Dirichlet boundary Γ_D. The spatial discretization is obtained by replacing the function u by $u_h \in \mathcal{S}^1_D$ in the weak form of the elliptic problem. We then prove the convergence of this discretization with respect to the diameters of the triangles $K \in \mathcal{T}$.

Finally, we provide the analytical and numerical tools for a finite element discretization of the following linear parabolic equation:

$$\begin{aligned} \frac{\partial u}{\partial t} - \operatorname{div}(D\nabla u) + \vec{c}\cdot\nabla u + ru &= f \quad \text{in } \Omega\times(0,T],\\ u &= 0 \quad \text{on } \Gamma_D\times(0,T],\\ \vec{n}\cdot D\nabla u &= g \quad \text{on } \Gamma_N\times(0,T],\\ u &= u_0 \quad \text{in } \Omega \text{ for } t = 0, \end{aligned} \tag{0.1}$$

with in general unknown solution u and data D, c, r, f, g, u_0. We stress that all of these quantities are real valued functions that may depend on space and time, e.g. $u(x,t) \in \mathbb{R}$, $D(x,t) \in \mathbb{R}^{2\times2}$ etc., whereas the initial condition u_0 depends only on x. The final time T is arbitrary, but kept fix in the following considerations. In order to have a unique solution, we assume that the data satisfy the following conditions[1]:

(P1) The coefficient functions satisfy $D \in \mathcal{C}(0,T;L^\infty(\Omega)^{2\times2})$, $\vec{c} \in \mathcal{C}(0,T;W^{1,\infty}(\Omega)^2)$, and $r \in \mathcal{C}(0,T;L^\infty(\Omega))$. Moreover, the data functions satisfy $f \in L^2(0,T;L^2(\Omega))$, $g \in L^2(0,T;L^2(\Gamma_N))$, and $u_0 \in H^1_D(\Omega)$.

[1] If $\|\cdot\|_{V(\Omega)}$ defines a norm on V, we use the space

$$\mathcal{C}(0,T;V(\Omega)) := \{v : (a,b) \to V \text{ measurable} \mid t \mapsto \|v(.,t)\|_V \text{ is continuous}\}.$$

For the definition of the other function spaces see Section 2.1.

(P2) $D(x,t) \in \mathbb{R}^2$ is symmetric and uniformly positive definite, i.e. there holds

$$\lambda := \inf_{0<t\leq T} \inf_{x\in\Omega} \min_{z\in\mathbb{R}^2\setminus\{0\}} \frac{z \cdot D(x,t)z}{|z|^2} > 0. \tag{0.2}$$

Furthermore, we assume that $D(x,t) \in \mathbb{R}^2$ is uniformly bounded, i.e. with

$$\kappa := \lambda^{-1} \sup_{0<t\leq T} \sup_{x\in\Omega} \max_{z\in\mathbb{R}^2\setminus\{0\}} \frac{z \cdot D(x,t)z}{|z|^2} \tag{0.3}$$

there holds $\kappa < \infty$. Note that $\kappa \geq 1$.

(P3) There is a constant $\beta \geq 0$, such that $r - \frac{1}{2}\operatorname{div} \vec{c} \geq \beta$ for almost all $x \in \Omega$ and $0 < t \leq T$.

(P4) There is a constant $c_r \geq 0$, such that $\|r\|_{L^\infty(0,T,L^\infty(\Omega))} \leq c_r\beta$.

(P5) On the Neumann boundary there is only outflow convection, i.e. there holds $\vec{c}(x,t)\cdot\vec{n}(x) \geq 0$ almost everywhere on Γ_N and for $0 < t \leq T$.

In Equation (0.1), the expression $\operatorname{div}(D\nabla u)$ is called the diffusion term, $\vec{c} \cdot \nabla u$ is called the convection term, and ru is called the reaction term. The function f on the right hand side is commonly denoted by source term or volume force. These terms can be understood with the following descriptions of specific cases of (0.1):

Diffusion dominated problems: If $\lambda \gg c_r\beta$ and $\lambda \gg c_{\vec{c}} := \|\vec{c}\|_{L^\infty(0,T;L^\infty(\Omega)^2)}$ the term $\operatorname{div}(D\nabla u)$ dominates (0.1). In physics diffusion means a process, in which the gradient ∇u is minimal. In this context, u may be a concentration or energy. In the first case, diffusion can be seen as the movement from a domain of high concentration to an area of lower concentration until the equilibrium is reached. The coefficient matrix D is proportional to the diffusivity, that is, it determines the speed and the direction of that process. The heat equation $\partial_t u - \triangle u = f$ is a well known example for this type of problems, c.f. Section 4.3.

Convection dominated problems: In the case of $c_{\vec{c}} \gg \lambda$ and $c_{\vec{c}} \gg c_r\beta$, the term $\vec{c} \cdot \nabla u$ determines the qualitative behavior of the solution u in (0.1). Convection or advection is the forced (active) transfer of concentration or energy. The coefficient vector $\vec{c}$ defines the direction as well as the speed of this process. We refer to Section 4.4 for the specification of an example, which is convection dominated.

Reaction dominated problems: The term ru dominates Equation (0.1) if $c_r\beta \gg \lambda$ and $c_r\beta \gg c_{\vec{c}}$. The name of this term is related to first-order reactions in chemistry. In contrast to zero-order reactions, where the reaction is independent of the concentration u, a first order reaction has a rate proportional to the concentration of one of the reactants. Thus, the coefficient r specifies the speed of this process. We refer to literature on physical chemistry for an example, e.g. Atkins [2, Sec. 27.3].

Note that the conditions $(P1)$–$(P5)$ admit all yet discussed regimes as well as mixed problem, where no term dominates. The lower bound λ from $(P2)$ for the smallest eigenvalue of the matrix D in time and space, determines the contribution of the gradient term to the energy norm

$$|\!|\!| v |\!|\!|_\Omega := \{\lambda\|\nabla v\|_\Omega^2 + \beta\|v\|_\Omega^2\}^{1/2}.$$

Analogously, the last term's contribution to the energy norm is quantified by the constant β from $(P3)$.
To derive the space-time discretization of (0.1), we take the standard approach: By integration by parts, we find the so-called weak form of the problem

$$(v\,;\partial_t u)+(\nabla v\,;D\nabla u)+(v\,;\vec{c}\cdot\nabla u)+(v\,;ru)=(v\,;f)+(v\,;g)_{\Gamma_N}\quad\forall v\in H^1_D(\Omega),$$

which is continuous in time for $t\in(0,T]$. For the time discretization, we choose $N-1$ time steps $t_0=0<t_1<\ldots<t_{N-1}<t_N=T$ between given time steps t_0,t_N. For spatial discretization, we replace $H^1_D(\Omega)$ by $\mathcal{S}^1_D(\mathcal{T}^{(n)})$, which consists of all $\mathcal{T}^{(n)}$-piecewise affine and globally continuous functions for $0\le n\le N$. Here, $\mathcal{T}^{(n)}$ is the spatial triangulation of the n^{th} time step. The time derivative $\partial_t u$ is approximated by the first order finite difference $(u_h^{(n)}-u_h^{(n-1)})/\tau_n$, where $u_h^{(n)}$ denotes the discrete solution at the n^{th} time step and τ_n is the time step size t_n-t_{n-1}. For the sequence of discrete solutions $\left(u_h^{(n)}\right)_{0\le n\le N}$, we denote by $u_{h,\tau}$ the function which equals $u_{h,\tau}(t_n)=u_h^{(n)}$ and which is locally affine with respect to time on all time intervals $[t_{n-1},t_n]$ with $0\le n\le N$. For the time discretization of the other entities, we choose a so-called θ-scheme, in which D is replaced by $D^{(n),\theta}:=\theta D(t_n)+(1-\theta)D(t_{n-1})$, as well as other data functions $\vec{c},r,f$ and g. This approach leads to N matrix equations with initial solution $u^{(0)}:=\Pi_{h,0}u_0$, where $\Pi_{h,0}$ denotes the L^2-projection onto the finite element space of the initial triangulation $\mathcal{T}^{(0)}$. Consequently, this provides a method to obtain an approximation for the solution u at the final time T. To answer the question if the method converges in time, we state a respective result from Verfürth [17], for data functions that do not depend on time and without convection.

At the end of this chapter, we briefly discuss the MATLAB function solving the involved matrix equation arising from the discretization. The matrix assembly is also specified.

In Chapter 3 we discuss the residual based a posteriori error estimator η. For simplicity of presentation, we restrict our attention to a simple model case and refer the reader to Section 3.7 for the precise statements in the general case. For constant data, backward Euler method(i.e. $\theta=1$), small convection, and without coarsening the error estimator η reads

$$\eta=\left\{\sum_{n=1}^{N}\tau_n\left(\eta^{(n)}\right)^2\right\}^{1/2}\text{ with }\eta^{(n)}=\left\{\left(\eta_S^{(n)}\right)^2+\left(\eta_\tau^{(n)}\right)^2\right\}^{1/2},$$

where the spatial and temporal estimators are defined by

$$\begin{aligned}\eta_S^{(n)}=&\left\{\sum_{K\in\mathcal{T}^{(n)}}\alpha_K^2\|R_K\|_K^2+\sum_{E\in\mathcal{E}^{(n)}}\lambda^{-1/2}\alpha_E\|R_E\|_E^2\right\}^{1/2},\\ \eta_\tau^{(n)}=&\|\!|u_h^{(n)}-u_h^{(n-1)}|\!\|_\Omega,\end{aligned}$$

respectively. By $\alpha_S=\min\{h_S\lambda^{-1/2},\beta^{-1/2}\}$, where $S\in\{K,E\}$, we denote the local parameter of an element or an edge, respectively. Furthermore, $\mathcal{E}^{(n)}$ is the set of all edges of $\mathcal{T}^{(n)}$. The spatial contribution $\eta_S^{(n)}$ includes the element residuals and the edge residuals, which are respectively given by

$$\begin{aligned}R_K=&\left[f-\frac{u_h^{(n)}-u_h^{(n-1)}}{\tau_n}-\vec{c}\cdot\nabla u_h^{(n)}-ru_h^{(n)}\right]\Big|_K,\\ R_E=&\begin{cases}-[\vec{n}\cdot D\nabla u_h^{(n)}]_E & \text{if } E\not\subseteq\Gamma,\\ g-\vec{n}\cdot D\nabla u_h^{(n)} & \text{if } E\subseteq\Gamma_N,\\ 0 & \text{if } E\subseteq\Gamma_D,\end{cases}\end{aligned}$$

where $[\cdot]_E$ is the edge jump over the edge E. Clearly, the above entities are more complicate in the general case. Spatially dependent data give rise to an additional term called data estimator, whereas time dependent data lead to contributions including time derivatives.
For the general problem with space-time dependent data, we show that the a posteriori error estimator η is equivalent to the error measured in the space-time energy norm

$$\|v\|_{X(a,b)} = \left\{ \|v\|^2_{L^\infty(a,b;L^2(\Omega))} + \|v\|^2_{L^2(a,b;H^1_D(\Omega))} + \int_a^b \|\partial_t v + \vec{c}(x,t)\cdot\nabla v\|^2_* dt \right\}^{1/2}.$$

First, we prove the equivalence of the residual

$$\begin{aligned}\langle R(u_{h,\tau})\,;\,v\rangle =& (f\,;\,v) + (g\,;\,v)_{\Gamma_N} - (\partial_t u_{h,\tau}\,;\,v) - (D\nabla u_{h,\tau}\,;\,\nabla v)\\ &- (\vec{c}\cdot\nabla u_{h,\tau}\,;\,v) - (r u_{h,\tau}\,;\,v) \quad \forall v\in H^1_D(\Omega) \text{ and } t\in(0,T].\end{aligned}$$

and the error. Afterwards, we decompose the residual in spatial and temporal contributions, which correspond to the respective estimators from above. For the spatial term, we show reliability and efficiency. Moreover, the temporal estimator is shown to be equivalent to the temporal contribution.

We also derive methods to obtain similar estimators for problems with large convection. The term $\|\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)} - u_h^{(n-1)})\|_*$ arising in the a posteriori analysis is then indispensable. In order to estimate this term an additional elliptic auxiliary problem has to be solved.

In adaptive methods, a common refinement $\widetilde{\mathcal{T}}^{(n)}$ of two regular triangulation $\mathcal{T}^{(n-1)}$, $\mathcal{T}^{(n)}$ from two adjacent time steps is required. Here, $\mathcal{T}^{(n)}$ is obtained from $\mathcal{T}^{(n-1)}$ by certain refinement and coarsening processes discussed in Chapter 1. It is shown, that the shape regularity constant $c_S(\widetilde{\mathcal{T}}^{(n)})$ of $\widetilde{\mathcal{T}}^{(n)}$ may be estimated by the shape regularity constant $c_S(\mathcal{T}^{(n-1)})$ of $\mathcal{T}^{(n-1)}$. Thus, with the results from Chapter 1, we can conclude that the shape regularity constants $c_S(\mathcal{T}^{(\ell)})$ of all triangulations of the solution process differs by factor of 4 from the shape regularity constant $c_S(\mathcal{T}^{(0)})$ of the initial triangulation $\mathcal{T}^{(0)}$. Furthermore, we introduce the MATLAB function computing the common refinement $\widetilde{\mathcal{T}}^{(n)}$.

At the end of this chapter, we discuss the implementation of the numerical algorithm for (0.1). The approach is adaptive in time and space and is controlled by a local tolerance parameter, which separately bounds the spatial and temporal contributions of $\eta^{(n)}$. Moreover, we describe the MATLAB functions that compute the error estimators $\eta^{(n)}$.

In Chapter 4 we apply the space-time adaptive strategy introduced above in numerical experiments. We verify the implemented MATLAB codes and visualize analytic conclusions of the preceding chapters. We compute the error in the energy norm and the error estimator η for two model problems with known analytic solution. The observed experimental convergence rates for a uniform and an adaptive strategy illustrate the asymptotic convergence rate $\mathcal{O}(\tau + h)$ predicted by the a priori and a posteriori analysis for the backward Euler scheme. Here, τ, h are the time step size and the mesh-width in space, respectively.

We discuss the heat equation $\partial_t u = \triangle u$ on an L-shaped domain with Dirichlet boundary data. In this case, the computations lead to suboptimal experimental convergence rates of the error estimator η.

Finally, we deal with a model describing pollution of the groundwater flow. The model specifications are introduced, as well as the physical correspondences to (0.1). We compare the discrete solution with an analogous analytic solution, to see if the algorithm exhibits the expected asymptotic behavior.

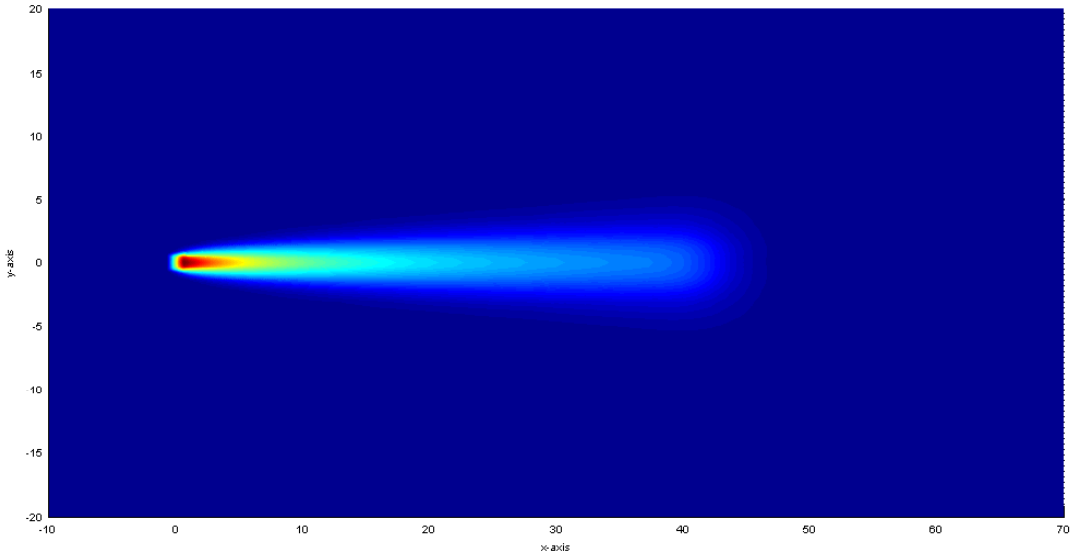


Figure 1: Plot of the discrete solution for a problem concerning the pollution in groundwater flow. The pollution source is at $(0, 0)$ and there is a groundwater flow in positive x direction. For a more detailed specification of the problem see Section 4.4.

The implementation has been carried out in MATLAB, Version 7.4.0287(R2007a). All subroutines can be found in the appendix.

Acknowledgment

I would like to thank Professor Dirk Praetorius and Professor Ewa Weinmüller for the supervision of my work. Their continuing support helped me to understand the underlying theory and the goals of the code implementation. Without ongoing encouragement of my parents and my girlfriend Eva the accomplishment of this thesis would have been impossible.

Vienna, July 2007
Philipp Wissgott

Chapter 1

Grid Management

In this section, we first introduce the entities required for the triangulation, and then discuss the refinement and coarsening strategy of the mesh. We also describe the grid management techniques leading to small and efficient data structures. At the end of this section, we specify the implemented refinement and coarsening functions.

1.1 Triangulation

Let Ω be a bounded two dimensional Lipschitz domain[1], see Figure 1.1, where the boundary $\Gamma := \partial\Omega$ satisfies $\Gamma = \overline{\Gamma}_D \cup \overline{\Gamma}_N$ with $\Gamma_N, \Gamma_D \subseteq \Gamma$ relatively open subsets and $\Gamma_D \cap \Gamma_N = \emptyset$. Γ_D is called **Dirichlet boundary** and is supposed to satisfy $|\Gamma_D| > 0$, where the surface measure is denoted by $|\cdot|$. Γ_N is called **Neumann boundary**. On Γ_D the solution satisfies Dirichlet conditions, assuming the values u_D, whereas on Γ_N the value of the generalized normal derivative is given. In order to apply the finite element method, we split Ω into a finite number of triangles.

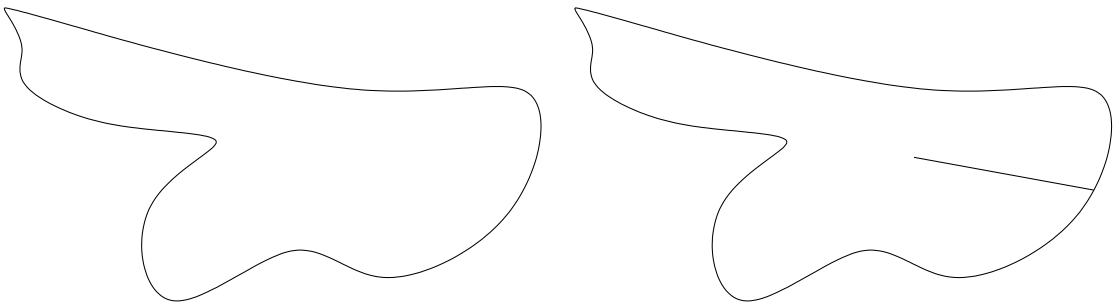
Definition 1. We call $\mathcal{T}$ a **triangulation** of Ω if

(i) $\mathcal{T}$ is a finite set of compact subsets of $\overline{\Omega}$,

(ii) $\overline{\Omega} = \bigcup \{T \,|\, T \in \mathcal{T}\}$, i.e. $\mathcal{T}$ is a covering of $\overline{\Omega}$,

(iii) $\forall\, T, T' \in \mathcal{T}$ with $T \neq T'$: $\operatorname{int} T \cap \operatorname{int} T' = \emptyset$, i.e. the interiors of two elements are disjoint,

[1] A Lipschitz domain is an open and connected domain with a compact boundary Γ which is locally parameterizable with Lipschitz-continuous functions. Furthermore, Ω is locally always on one side of Γ.



(a) (b)

Figure 1.1: (a) Example of a Lipschitz domain. (b) Example of a non-Lipschitz domain.

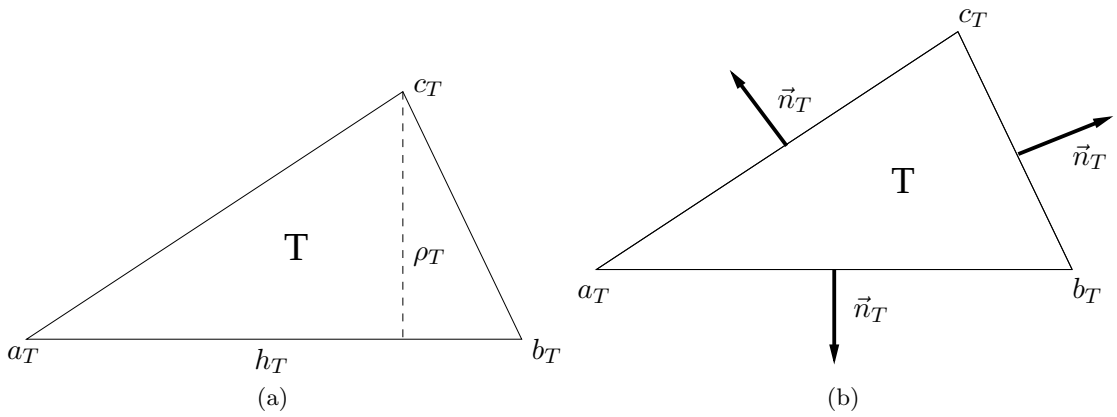


Figure 1.2: (a) Notations for an element T. (b) Normal vectors $\vec{n}_T$ for an element T.

(iv) $\forall\, T \in \mathcal{T}:\ T \cap \Gamma \in \{T \cap \overline{\Gamma}_D, T \cap \overline{\Gamma}_N, \emptyset\}$, i.e. there is no element which intersects Dirichlet boundary Γ_D and Neumann boundary Γ_N.

Furthermore we restrict our attention to the case, where all elements $T \in \mathcal{T}$ are non-degenerate triangles, i.e. there exist non-collinear **vertices** $a, b, c \in \overline{\Omega}$ such that $T = \mathrm{conv}\{a, b, c\}$. For each triangle $T \in \mathcal{T}$, we denote these vertices by $a_T, b_T, c_T \in \overline{\Omega}$. The choice of a_T is arbitrary and b_T, c_T are chosen counter-clockwise, c.f. Figure 1.2(a).
Moreover, we denote the length of the longest edge of T by $h_T > 0$ and by $\rho_T > 0$, we denote the height on this edge. In particular, $|T| = h_T \rho_T / 2 > 0$ holds.

Remark. Definition 1 (ii) implies that $\overline{\Omega}$ has a polygonal boundary, in other words we neglect the approximation of Ω.

Definition 2. For a triangulation $\mathcal{T}$, we denote by

$$\mathcal{N} := \big\{ x \in \overline{\Omega} \,\big|\, \exists\, T \in \mathcal{T} :\ x \text{ is a vertex of } T \big\}$$

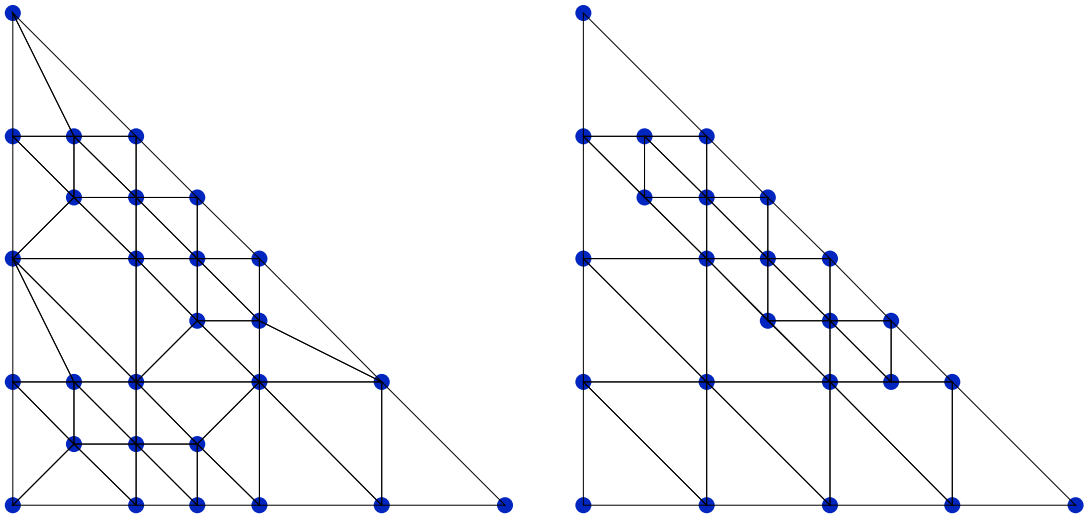
the **set of all nodes** of $\mathcal{T}$. By

$$\mathcal{N}_D := \big\{ x \in \mathcal{N} \,\big|\, x \in \overline{\Gamma}_D \big\}$$

we denote the **set of all Dirichlet nodes**.

Definition 3. In a triangulation $\mathcal{T}$ a compact subset $E \subseteq \overline{\Omega}$ with $|E| > 1$ is called **edge**, if there exist two elements $T, T' \in \mathcal{T}$ such that $E = T \cap T'$, or if there exists a $T \in \mathcal{T}$ such that $E = T \cap \Gamma = \mathrm{conv}\{a, b\}$ with $a, b \in \Gamma$. In the first case, T and T' are called **neighboring elements**, in the latter E is called a **boundary edge**. The **set of all edges** of $\mathcal{T}$ is denoted by $\mathcal{E}$, whereas the subset of $\mathcal{E}$ of all boundary edges is denoted by $\mathcal{E}_\Gamma := \big\{ E \in \mathcal{E} \,\big|\, E \subseteq \Gamma \big\}$. Analogously to the vertices of an element, we denote the vertices of an edge by $\{a_E, b_E\}$, that is $E = \mathrm{conv}\{a_E, b_E\}$. If the sequence of the vertices a_E and b_E is relevant, we indicate this by E_{a_E, b_E} which is the **oriented edge** from a_E to b_E. E_{b_E, a_E} is the oriented edge from b_E to a_E.

For a triangle $T \in \mathcal{T}$, we denote by $\vec{n}_T$ the outer **normal vector** on the boundary ∂T with respect to T, c.f. Figure 1.2(b). Analogously, we denote by $\vec{n}_\Gamma(x)$ the outer normal vector on the boundary Γ. Note that $\vec{n}_\Gamma(x)|_{\partial T \setminus \{a_T, b_T, c_T\}} := \vec{n}_T$ for $x \in \Gamma$ and $T \in \mathcal{T}$ with $\partial T \cap \Gamma \neq \emptyset$. In particular, the normal vector $\vec{n}_\Gamma(x)$ is defined almost everywhere, namely at least on $\Gamma \setminus \mathcal{N}$.



(a) (b)

Figure 1.3: (a) Example of a regular triangulation. (b) Example of a grid.

Definition 4. Let $\mathcal{T}$ be a triangulation of Ω. A node $x \in \mathcal{N}$ is a **hanging node**, if there exists an element $T \in \mathcal{T}$ such that $x \in T \setminus \{a_T, b_T, c_T\}$, i.e. x is a vertex of a neighbor of T without being a vertex of T. The **set of all hanging nodes** is denoted by $\mathcal{N}_h$. Note that $\mathcal{N}_D \cap \mathcal{N}_h = \emptyset$. The **set of all free nodes** $\mathcal{N}_f$ is defined by $\mathcal{N}_f := \mathcal{N} \setminus (\mathcal{N}_D \cup \mathcal{N}_h)$.
A triangulation $\mathcal{T}$ with $\mathcal{N}_h = \emptyset$ is **regular** in the sense of Ciarlet, c.f. Carstensen [7] and Figure 1.3(a). In the code implementation we will allow a certain kind of hanging nodes.

Definition 5. We call a triangulation $\mathcal{G}$ a **grid**, if there is at most one hanging node per triangle, i.e. $|(T \setminus \{a_T, b_T, c_T\}) \cap \mathcal{N}_h| \leq 1, \ \forall T \in \mathcal{G}$, c.f. Figure 1.3(b). This condition is called the **grid condition**.

Note that this definition allows nested hanging nodes, see nodes 8 and 16 in Figure 1.4(a), since they belong to two different triangles but it does not allow multiple hanging nodes for one element, c.f. nodes 18 and 19 in Figure 1.4(b).

1.1.1 Patches

For the a posteriori error analysis we need the so-called patches of nodes, edges, and elements.

Definition 6. For a regular triangulation $\mathcal{T}$, we denote by

$$\omega_a := \bigcup \{T \in \mathcal{T} \,|\, a \cap T \neq \emptyset\}$$

the **patch of a node** $a \in \mathcal{N}$, c.f. Figure 1.5(a), by

$$\omega_E := \bigcup \{T \in \mathcal{T} \,|\, E \subseteq T\}$$

the **patch of an edge** $E \in \mathcal{E}$, c.f. Figure 1.5(b), and by

$$\omega_T := \bigcup \{T' \in \mathcal{T} \,|\, T \cap T' \neq \emptyset\}$$

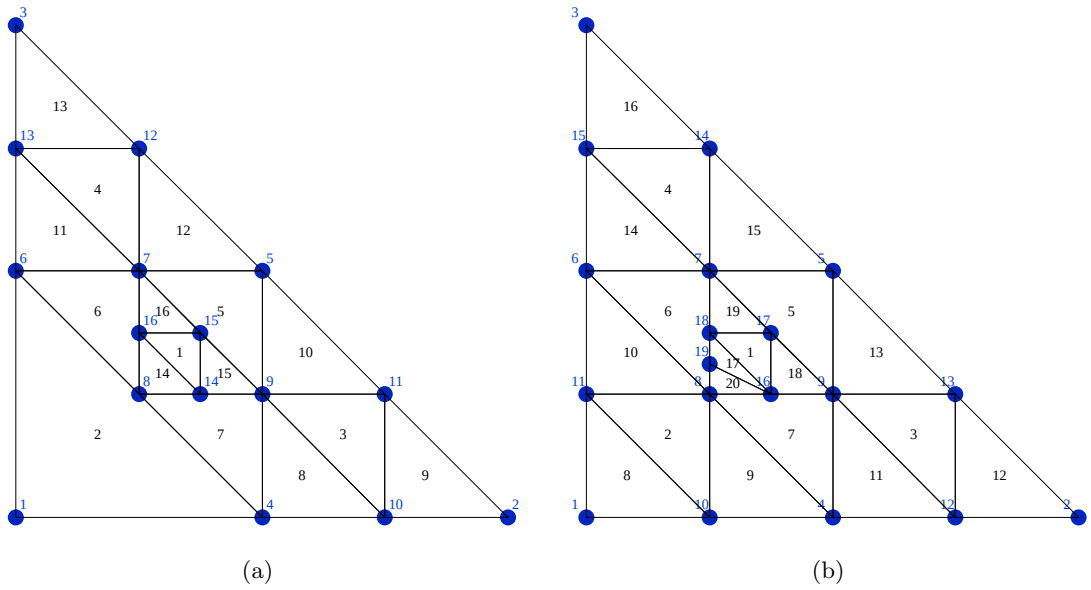


Figure 1.4: (a) Example of a grid with nested hanging nodes: Element 6 has one hanging node 16. Node 8 is a hanging node for element 2, but not for element 6, since node 8 is one of its vertices. Note that the element 14 has no hanging nodes, though it includes the vertices 8 and 16. (b) Example of an irregular triangulation which is not a grid: For the element 6 there are two hanging nodes, 18 and 19.

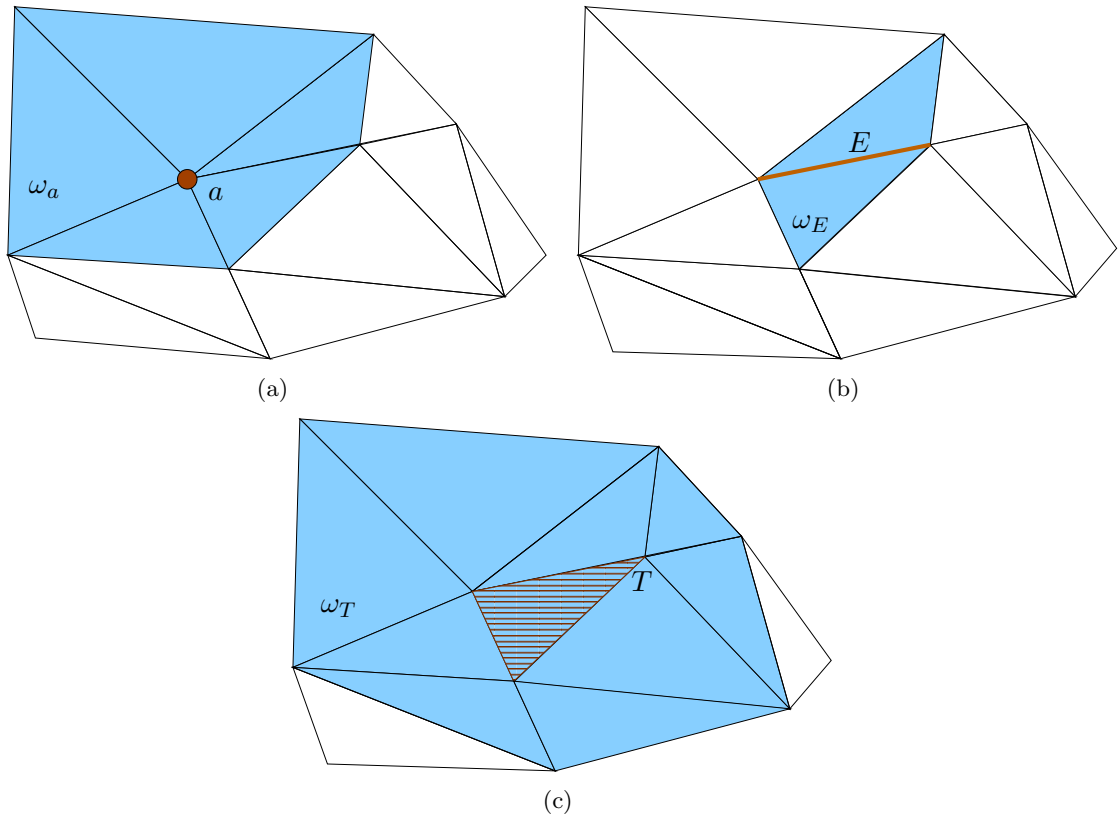


Figure 1.5: (a) Patch of a node a. (b) Patch of an edge E. (c) Patch of an element T.

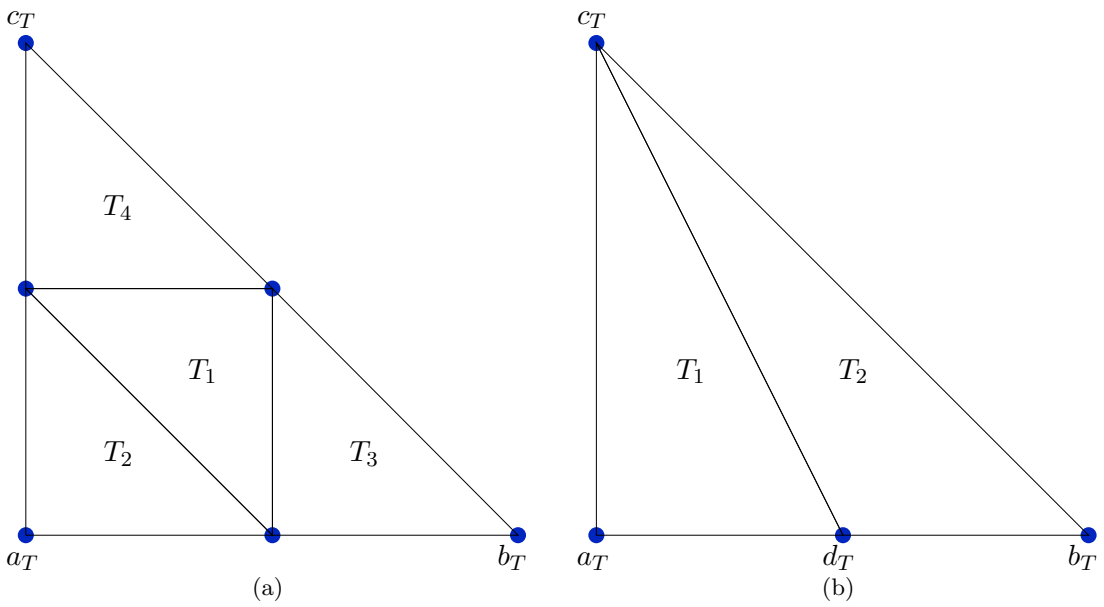


Figure 1.6: (a) Red refinement with the children T_1, T_2, T_3 and T_4. (b) Green refinement with the green brothers T_1 and T_2.

the **patch of an element** $T \in \mathcal{T}$, c.f. Figure 1.5(c).

1.2 Refinement and Coarsening

In the following, let $\mathcal{T}_0$ be an initial regular triangulation of the domain Ω. To decrease the error, one usually has to refine this initial triangulation. However, in time dependent problems, it may pay to undo certain refinements, or with other words to coarsen the grid. This results in lower computational complexity, especially when the solution becomes smoother which is sometimes the case for standard parabolic problems. While performing these grid manipulations, one should try to follow the rules listed below.

(I) Keep regularity of the grid unchanged.

(II) While refining the grid, keep in mind that you possibly have to undo this process.

(III) Avoid almost degenerated triangles, where the quotient $h_T^2/|T|$ is large, since $\max_{T\in\mathcal{T}}(h_T^2/|T|)$ affects the constants in the a priori and a posteriori error estimates.

(IV) Try to keep data structures simple and small.

First we specify the details of the **refinement strategy**:

CONVENTION. We first formulate the local grid-refinement rules for a triangle T of a grid $\mathcal{G}$ which are admitted while constructing a refined triangulation $\mathcal{T}$:

(i) **Red-refinement:** The element $T \in \mathcal{G}$ is split into 4 similar triangles $T_1, \ldots, T_4 \in \mathcal{T}$, such that $T_1 \cup T_2 \cup T_3 \cup T_4 = T$, c.f. Figure 1.6(a).

(ii) **Green-refinement:** The element $T \in \mathcal{G}$ is split into 2 triangles $T_1, T_2 \in \mathcal{T}$ such that $T_1 := \text{conv}\{d_T, c_T, a_T\}$ and $T_2 := \text{conv}\{d_T, b_T, c_T\}$, where $d_T := (a_T + b_T)/2$ is the barycenter of an edge of T, c.f. Figure 1.6(b).

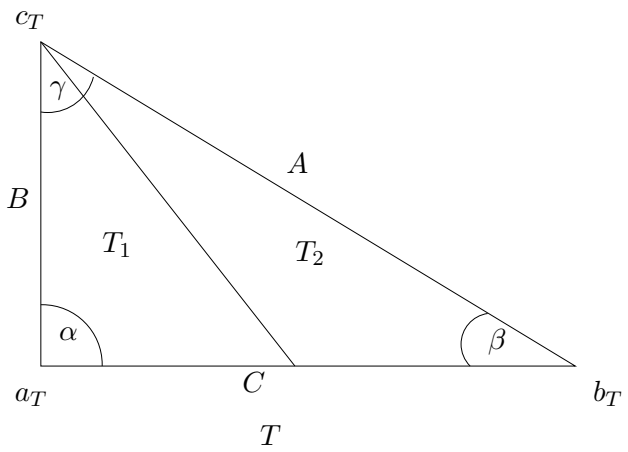


Figure 1.7: Notations for the proof of Lemma 1.1.

Definition 7. Let $\mathcal{G}$ be a triangulations and assume that $\mathcal{T}$ was obtained from certain refinements of elements of $\mathcal{G}$. If $T \in \mathcal{G}$ is red-refined, we call T the **father** and $T_i \in \mathcal{T}$, $i = 1, \dots, 4$, his **children**. Furthermore, we call the child which has no common vertices with his father the **middle-child**. The middle child is always denoted by T_1, c.f. Figure 1.6(a). Analogously, we call $T \in \mathcal{G}$ **twin** if T is green-refined, T_1 and T_2 are the **green brothers**, c.f. Figure 1.6(b). Additionally, $T_1 \in \mathcal{T}$ is called the **firstborn**, which is always to the left of the new edge, if seen from d_T to c_T. $T_2 \in \mathcal{T}$ is called the **lastborn**. Finally, we denote fathers whose children have no children themselves by **last generation fathers**.

The vertices of a middle-child lie on the barycenter of the edges of its father $T \in \mathcal{G}$. Thus, on a neighboring element $T' \in \mathcal{G}$ of T red-refinement leads to a hanging node on the barycenter of the edge which is shared by T and T', if T' is not red-refined as well.

Lemma 1.1. *(i) For the triangles T_1,...,T_4 obtained from a red-refinement of $T \in \mathcal{G}$, $h_{T_i} = h_T/2$ and $|T_i| = |T|/4$ for $i = 1, \dots, 4$ holds. In particular, $h_{T_i}^2/|T_i| = h_T^2/|T|$, i.e. the shape regularity is not affected.*
(ii) For the triangles T_1 and T_2 obtained from a green-refinement of $T \in \mathcal{G}$, there holds $h_{T_i}^2/|T_i| \leq 2h_T^2/|T|$ and $|T_i| = |T|/2$ for $i = 1, 2$, that is the shape regularity constant increases at most by a factor 2.

Proof.(i) is obvious because of the similarity of the children of T. (ii) Consider the triangle T with the notation from Figure 1.7. Elementary geometric considerations yield

$$|T| = \frac{|c_T - a_T||b_T - a_T|}{2} \sin(\alpha),$$

c.f. [18, §3.2, p.759]. In particular, this means that

$$|T_1| = \frac{1}{2}\left(|c_T - a_T|\frac{|b_T - a_T|}{2}\right)\sin(\alpha) = \frac{|T|}{2}$$

and $|T_2| = |T| - |T_1| = |T|/2$. Moreover, $h_{T_i} \leq h_T$, hence $h_{T_i}^2/|T_i| \leq 2h_T^2/|T|$. ■

For **coarsening** the red-refinement process is locally undone.

CONVENTION. Let $\mathcal{G}$ be a grid obtained by red-refinement of the initial triangulation $\mathcal{T}_0$. Then, we allow the following local grid-coarsening rule for the triangles $T_1, \dots, T_4 \in \mathcal{G}$, to obtain a coarsened triangulation $\mathcal{T}$:

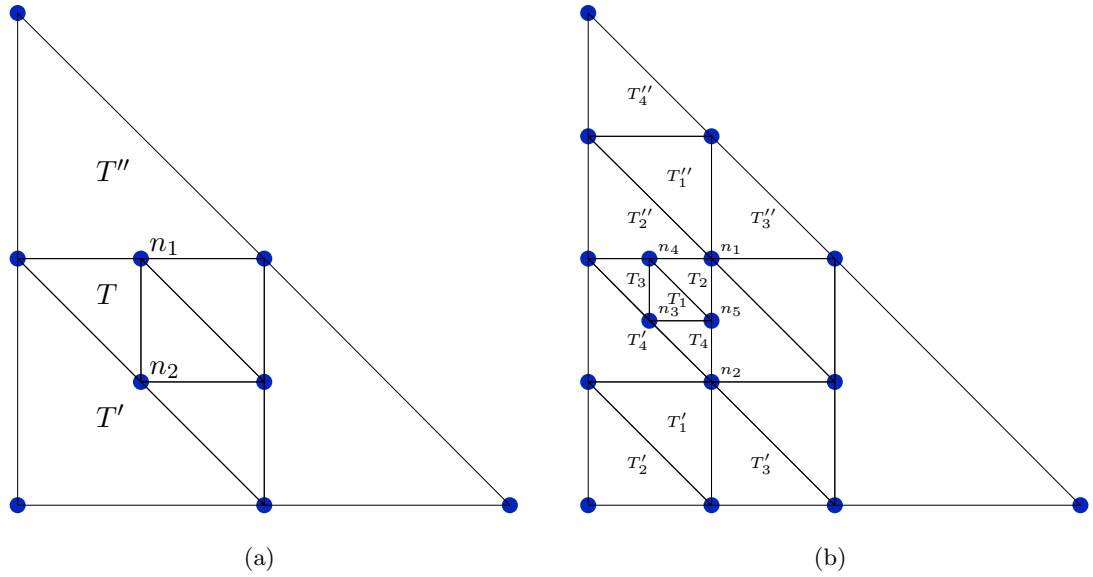


Figure 1.8: Offensive refinement: (a) Red-refinement of element T would create two hanging nodes for the elements T' and T'' since n_1 and n_2 are already hanging nodes. (b) Thus, these elements are red-refined and we obtain $T_1', \ldots, T_4'$ and $T_1'', \ldots, T_4''$. Then T is red-refined and we obtain $T_1, \ldots, T_4$. Note, that n_1 and n_2 are no longer hanging nodes, but n_3 and n_4 are, for T_4' and T_2'', respectively.

Red-coarsening: The 4 children $T_1, \ldots, T_4 \in \mathcal{T}$ are reunited to their father $T \in \mathcal{T}$, such that $T_1 \cup T_2 \cup T_3 \cup T_4 = T$.

We allow only red-refinement and red-coarsening in grid management. Red-refining all elements of a regular triangulation $\mathcal{T}$ is called **uniform refinement**. Usually, this leads to a certain inefficiency. On the other hand an **adaptive refinement** which only red-refines certain elements, leads to hanging nodes in the resulting triangulation $\mathcal{T}'$. In this case we red-refine all elements with more than one hanging node to obtain a grid $\mathcal{G}$ (this process may also lead to a uniform refinement). We call this procedure **grid closure**. By green-refinement of all elements with one hanging node, we obtain the regular triangulation $\overline{\mathcal{G}}$ from $\mathcal{G}$. This last step is called **green closure**[2] of the grid $\mathcal{G}$.
For a sequence of time steps $t_0, \ldots, t_N$, we denote the corresponding **sequence of grids** by $\left(\mathcal{G}^{(k)}\right)$, $k = 0, \ldots, N$. With the following grid management strategy, a grid $\mathcal{G}^{(k)}$ is conveyed to its successor grid $\mathcal{G}^{(k+1)}$ with $\mathcal{G}^{(0)} := \mathcal{T}_0$.

STRATEGY. Let $\left(\mathcal{G}^{(k)}\right)$ be a sequence of grids. By $\left(\mathcal{T}_1^{(k)}\right)$ and $\left(\mathcal{T}_2^{(k)}\right)$ we denote the corresponding sequences of triangulations, which may violate the grid condition. Then, the **grid management strategy** is as follows:

1. Compute the discrete solution for $\overline{\mathcal{G}}^{(k)}$.
2. An error indicator decides whether to stop or to continue the computation.
3. An element estimator decides whether an element $T \in \mathcal{G}^{(k)}$ is marked for red-refinement or -coarsening.
4. Red-refine all marked elements of $\mathcal{G}^{(k)}$ to obtain $\mathcal{T}_1^{(k+1)}$. The refinement is offensive, that is all elements of $\mathcal{G}^{(k)}$ which are marked are red-refined. If red-refinement of an element

[2] It is also possible to keep hanging nodes in the grid, but then one has to impose additional auxiliary conditions.

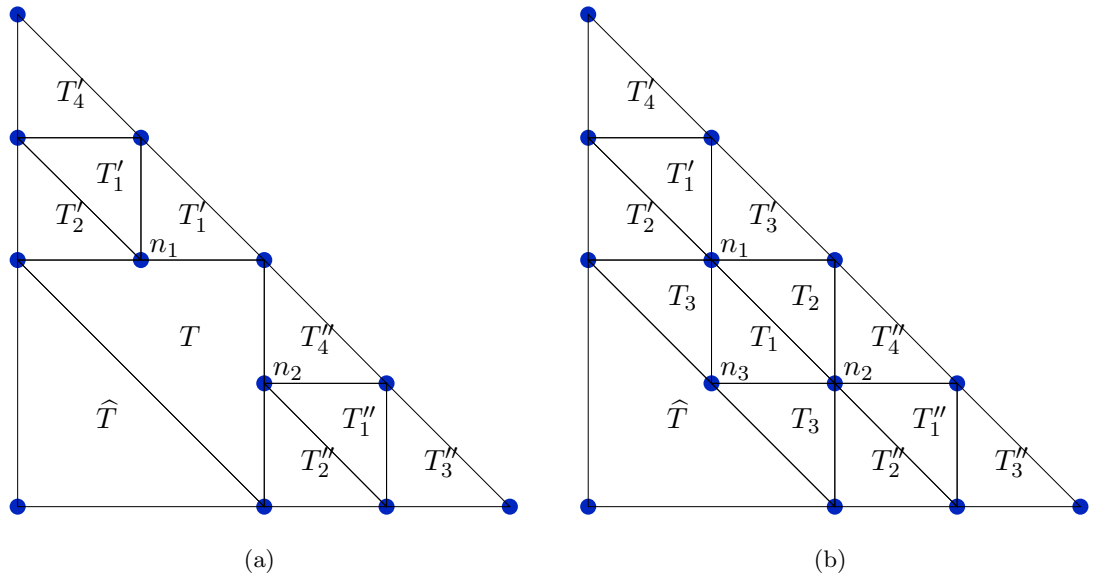


Figure 1.9: One hanging node per triangle: (a) T' and T'' were marked for refinement and red-refinement created two hanging nodes n_1 and n_2 for T. (b) Thus, T is also red-refined, though it was not marked. The node n_3 is the only hanging node for $\widehat{T}$, so no red-refinement is done.

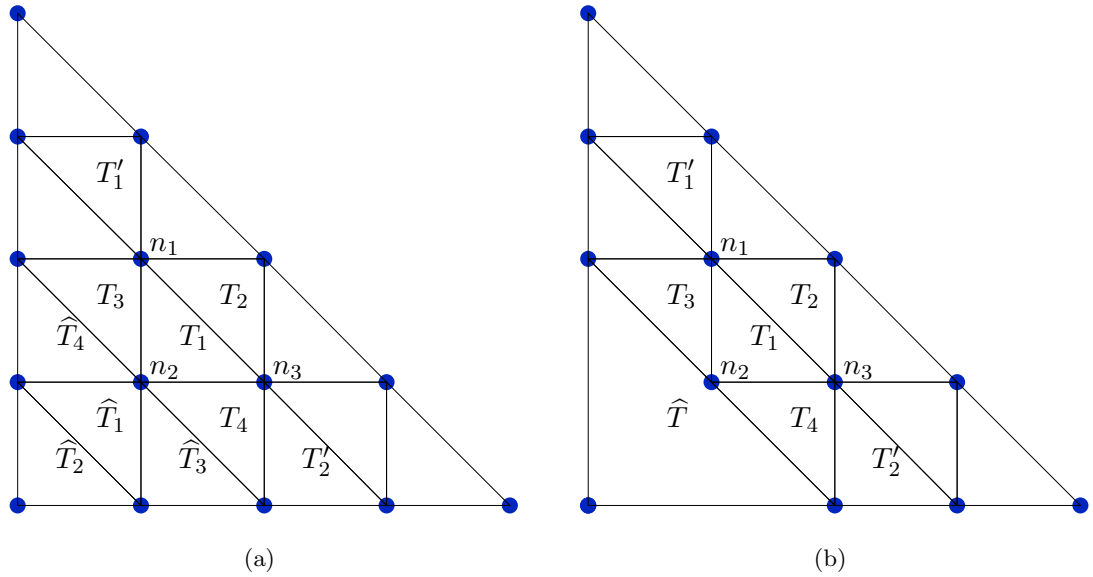
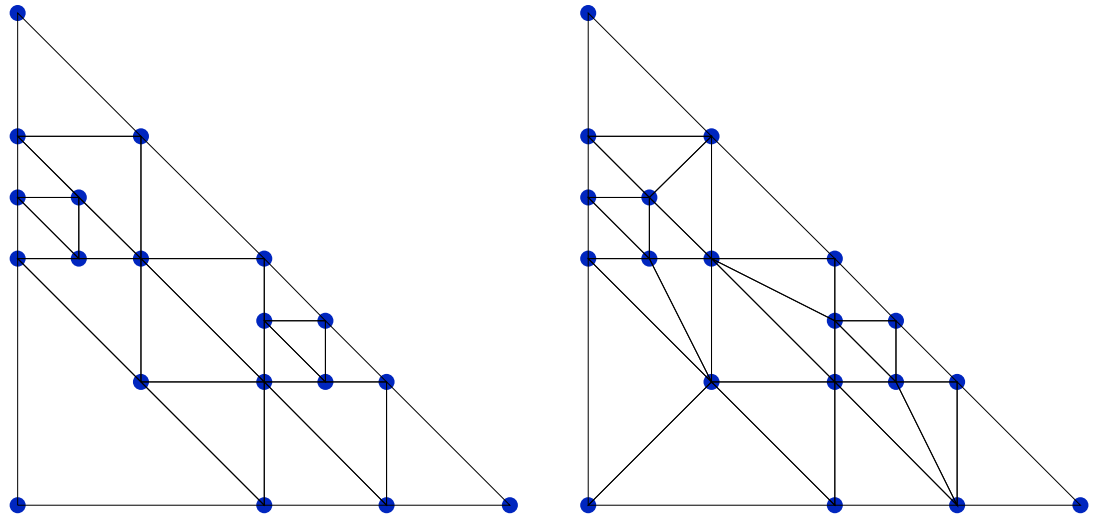


Figure 1.10: Defensive coarsening: (a) Consider the case that all elements but T_1' and T_2' are marked for coarsening. Thus, we do not coarsen T_1', T_2' and the other children of their father. Moreover we do not coarsen $T_1, \ldots, T_4$ although they are all marked, since red-coarsening would create two hanging nodes n_1 and n_3 for their father T. (b) The elements $\widehat{T}_1, \ldots, \widehat{T}_4$ were red-coarsened to their father $\widehat{T}$, because they were all marked and there is just one hanging node n_2.



(a) (b)

Figure 1.11: Green closure: Grid before (a) and after (b) green-closure of all elements with a hanging node.

$T \in \mathcal{G}^{(k)}$ would result in two hanging nodes on one edge of another element $T' \in \mathcal{G}^{(k)}$, the element T' is red-refined first and then the element T is red-refined, c.f. Figure 1.8.

5. Perform a grid closure on the triangulation $\mathcal{T}_1^{(k+1)}$ to obtain the temporary grid $\mathcal{G}_1^{(k+1)}$, c.f. Figure 1.9(b).

6. Coarsening of $\mathcal{G}_1^{(k+1)}$ to obtain $\mathcal{T}_2^{(k+1)}$. The coarsening strategy is defensive, i.e. the algorithm only coarsens if all children of a father are marked for coarsening and afterwards refines until the resulting triangulation is a grid, c.f. Figure 1.10. Note, that we only include last generation fathers in this process.

7. Perform a grid closure on the triangulation $\mathcal{T}_2^{(k+1)}$ to obtain the grid $\mathcal{G}^{(k+1)}$.

8. Update $k \mapsto k+1$ and return to step (1).

Remark. For coarsening, we simply reverse the refinements triangle per triangle. However, it would be possible to proceed in another manner, e.g. mark an entire polygon for coarsening and fill it with a minimal number of triangles. The disadvantage of the latter method is that the shapes of the elements of a triangulation may change.

STRATEGY. The discrete solution is computed on a regular triangulation $\overline{\mathcal{G}}^{(k)}$ obtained by a **green closure** of the grid $\mathcal{G}^{(k)}$. The error indicator marks a twin $T \in \mathcal{G}^{(k)}$ for red-refinement if at least one of its green brothers $T_1, T_2 \in \overline{\mathcal{G}}^{(k)}$ is marked for refinement. It marks a twin $T \in \mathcal{G}^{(k)}$ red-coarsening if both green brothers are marked for coarsening.

At the beginning of this section we required certain conditions to be satisfied during the grid adaptation. The above strategy keeps the regularity and uses a "nested" structure where refinements can easily be undone. Furthermore, Lemma 1.1 shows that the shapes of the elements of the initial grid $\mathcal{G}^{(0)}$ remain almost unchanged: For all $T \in \mathcal{G}^{(k)}$ and $0 \leq k \leq N$ with

$T \subseteq T_0 \in \mathcal{G}^{(0)}$

$$\frac{h_T^2}{|T|} \leq 2\frac{h_{T_0}^2}{|T_0|}. \tag{1.1}$$

holds. Thus, the third condition, see III page 13, is satisfied. The fourth condition IV will be discussed in the following section.

1.3 Data structures and Code for Triangulation

In the implementation of the code, a given grid $\mathcal{G}^{(k)}$ is stored in the data structure $Grid(k)$, which contains several sub-arrays. The index (k) indicates that all entities are related to the grid $\mathcal{G}^{(k)}$.

Grid(k).C4N: Abbreviation for "coordinates for nodes". For $n = \#\mathcal{N}^{(k)}$, $Grid(k).C4N$ is a $n \times 2$ array, where the i^{th} row corresponds to the coordinates of the i^{th} node, i.e. a node $z_i = (x, y) \in \mathcal{N}$ is stored in $Grid(k).C4N = [x\ y]$.
Format: x-coordinate [double] y-coordinate [double]

Grid(k).N4E: Abbreviation for "nodes for elements". For $n = \#\mathcal{G}^{(k)}$, Grid(k).N4E is a $n \times 5$ array, where the i^{th} row corresponds to the i^{th} element T_i of $\mathcal{G}$. For this element, the first three columns correspond to the rows of $Grid(k).C4N$ and store the vertices of T_i, that is if ℓ,m and p are the rows of the vertices in $Grid(k).C4N$, than $Grid(k).N4E(i, 1:3) = [\ell\ m\ p]$ holds. If T_i has a father F_i, the fourth column corresponds to the row of $Grid(k).F4E$ where F_i is stored. If T_i has a green brother T_j, the fifth column contains $Grid(k).N4E(i, 5) = j$ and the green brother itself is stored at the j^{th} row of $Grid(k).N4E$. If T_i is red-refined, it becomes a father, see $Grid(k).F4E$. Its middle-child replaces T_i in $Grid(k).N4E$. Moreover there may be new vertices, which are stored in $Grid(k).C4N$. If T_i is green-refined the firstborn takes its place in $Grid(k).N4E$. For coarsening the corresponding processes are reversed. Note, that after refinement T_i is no longer stored in $Grid(k).N4E$.

Format(col. 1-3):	ℓ [integer]	m [integer]	p [integer]
	($\rightarrow$ Grid(k).C4N)	($\rightarrow Grid(k).C4N$)	($\rightarrow Grid(k).C4N$)
Format(col. 4-5)	F_i [integer]	j [integer]	
	($\rightarrow Grid(k).F4E$)	($\rightarrow Grid(k).N4E$)	

Grid(k).F4E: Abbreviation for "fathers for elements". If $F \in \mathcal{G}^{(k)}$ is red-refined, its vertices and children are stored in $Grid(k).F4E$. For $n = (\#$ of fathers in $\mathcal{G}^{(k)})$, Grid(k).F4E is a $n \times 8$ array, where the i^{th} row corresponds to the i^{th} father F_i of $\mathcal{G}^{(k)}$. For this father, the first three columns correspond to the rows of $Grid(k).C4N$ and store the vertices of F_i, i.e. if ℓ,m and p are the rows of the vertices in $Grid(k).C4N$, $Grid(k).F4E(i, 1:3) = [\ell\ m\ p]$. The fourth to seventh column correspond to $Grid(k).N4E$ and store the rows of the children of F_i, i.e. if q,r,s and t are the rows of the children in $Grid(k).N4E$, the columns contain $Grid(k).F4E(i, 4:7) = [q\ r\ s\ t]$. Note that the middle child is always stored in the fourth column. If F_i itself has a father F_j, the eighth column stores j.

Format(col. 1-3):	ℓ [integer]	m [integer]	p [integer]
	($\rightarrow Grid(k).C4N$)	($\rightarrow Grid(k).C4N$)	($\rightarrow Grid(k).C4N$)
Format(col. 4-6)	q [integer]	r [integer]	s [integer]
	($\rightarrow Grid(k).N4E$)	($\rightarrow Grid(k).N4E$)	($\rightarrow Grid(k).N4E$)
Format(col. 7-8)	t [integer]	j [integer]	
	($\rightarrow Grid(k).N4E$)	($\rightarrow Grid(k).F4E$)	

Grid(k).NOE: Abbreviation for "nodes on edges". For $n = \#\mathcal{N}^{(k)}$ $Grid(k).NOE$ is a $n \times n$ matrix. Consider the notations of Figure 1.12(a) and let T be the red-refined element with the vertices $n_1, n_2, n_3 \in \mathcal{N}^{(k)}$, which are stored in the first three rows of $Grid(k).C4N$. Furthermore, let its middle child T_1 have the vertices $n_4, n_5, n_6 \in \mathcal{N}^{(k+1)}$ which again correspond to the rows of $Grid(k).C4N$. Then, the $(1,2)$ entry contains $Grid(k).NOE(1,2) = n_4$ if n_4 is no boundary node. If n_4 is a boundary node the entry is 0. Generally, all none-boundary nodes which lie on the edge between two nodes are stored in $Grid(k).NOE$. Note, that the matrix is not symmetric, since it stores the nodes of a middle child, such that only the pairs of vertices of the father in mathematically positive order are considered in Figure 1.12(a) $(1,2), (2,3)$ and $(3,1)$. On the other hand, the $(2,1), (3,2)$ and $(1,3)$ entries of $Grid(k).NOE$ are trivial. For a more complicated example see Figure 1.12(b).
Format $Grid(k).NOE(i,j) = m$ [integer]

Grid(k).Unused: Necessary for hygiene of the data structures. Classified in $Unused.C4N$, $Unused.N4E$ and $Unused.F4E$ each field being a vertical vector with the integers of currently unneeded rows referring to the adequate field. If an element becomes red-coarsened three entries of $Grid(k).N4E$ are deleted, since the former father is stored in the row of the middle child, which does not exist anymore. Moreover the father is deleted from $Grid(k).F4E$ and the vertices of the middle child which are not shared by another element are deleted from $Grid(k).C4N$. All rows of these former entries are stored in the corresponding $Grid(k).Unused$ fields and are reused if new refinement processes are conducted.

Grid(k).M,Grid(k).H,Grid(k).b: To enhance the performance we store all relevant data obtained in the course of computations. More precisely, we store the mass matrix in the sub-array M, the transport matrix in the field H, and the right hand side in the vector b, see (2.32) below.

Remark. If there is no node, element, father or green brother the default value is 0.

Note that a child can be refined again, while the data in $Grid(k).F4E$ remains unchanged. Only for last generation fathers the information in this field directly corresponds to the actual grid. Hanging nodes can easily be identified by subtracting $Grid(k).NOE - Grid(k).NOE' := M$. Only hanging nodes have no counterpart and thus remain as the nonzero entries of M.
Wherever possible we use sparse matrices, e.g. for NOE. Prelocating the memory is particularly useful when working with large versions of these matrices. This is done, by storing arbitrary values at the location, such that during the storage process this value is simply exchanged by the right one. Otherwise the location of new memory would result in bad performance.

1.3.1 The Function $refineGrid$

After reversing a possible green completion, the function $refineGrid$ red-refines all elements of a given $Grid$. Furthermore, it includes a *while*–loop which ensures the grid condition by red-refinement of those elements, which have more than one hanging node after the previous refinement processes. Finally, we obtain a regular triangulation by a green closure.
In many parts of the implementation, we use the build-in MATLAB function

$$A = \text{sparse}(IndexVector1, IndexVector2, Values, m, k)$$

for matrix assembly. This method highly increases the speed of computations. In this case, $A \in \mathbb{R}^{m \times k}$ is a $m \times k$ matrix, whose entries are given as follows: With $i = IndexVector1(\ell)$

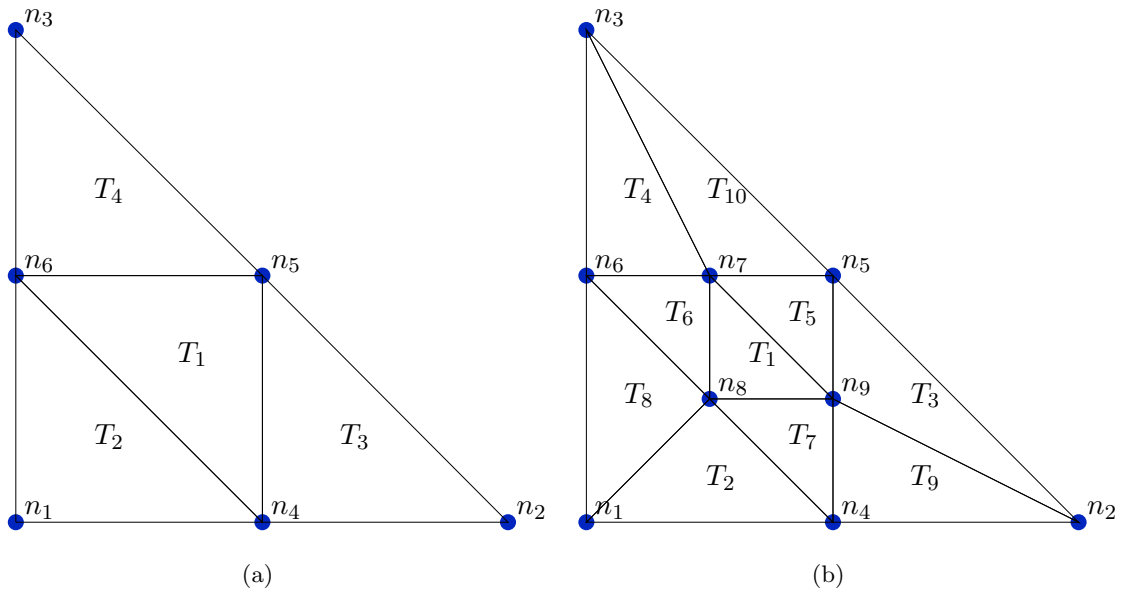


Figure 1.12: Primitive (a) and more complicated example (b) for $Grid(k).NOE$. In (b) $Grid(k).NOE(1,2) = Grid(k).NOE(2,3) = Grid(k).NOE(3,1) = 0$ since the corresponding nodes n_4,n_5 and n_6 are boundary nodes. $Grid(k).NOE(4,5) = 9$ but $Grid(k).NOE(5,4) = 0$ because n_9 is a node on a edge connecting n_4 and n_5 but also a hanging node for the triangle $\{n_5, n_4, n_2\}$. Accordingly, $Grid(k).NOE(5,6) = 7$ and $Grid(k).NOE(6,4) = 8$ and all other entries of $Grid(k).NOE$ are trivial.

and $j = IndexVector2(\ell)$ holds $A_{ij} = Values(\ell)$. The remaining entries are zero. Fortunately, values from multiple index entries are automatically added to the former value.
The actual red-refinement is done by the subfunction $refineElements$. The first column of the input matrix $MarkedElements$ indicates which elements are to be red-refined. All lines in this column, which are not trivial, correspond to a marked element in $Grid.N4E$ (abbreviated by $N4E$). Thus,

```
0213        nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0214        mnze = intersect(MarkedElements,nze); %marked non zero elements
```

gives the index vector $mnze$, abbreviating marked non zero elements. Each red-refinement creates new nodes, which may violate the grid condition. Thus, after a refinement process we have to check the remaining elements and count their hanging nodes. To minimize the number of elements which have to be checked, we restrict to elements and fathers, which are neighbors of the refined triangles.

```
0218        a = [N4E(nze,1) N4E(nze,2) N4E(nze,3)]';
0219        I1 = reshape(a,3*nnz(nze),1);
0220        a = [N4E(nze,2) N4E(nze,3) N4E(nze,1)]';
0221        I2 = reshape(a,3*nnz(nze),1);
0222        a = [nze; nze; nze];
0223        Values = reshape(a,3*nnz(nze),1);
0224        ElementsOverEdge = sparse(I1,I2,Values,size(C4N,1),size(C4N,1));
0225        ElementsToCheck = zeros(1,6*nnz(mnze));
0226        for j=1:size(mnze,2)
0227            Nodes = N4E(mnze(j),1:3);
0228            if N4E(mnze(j),4)
0229               FatherNodes = F4E(N4E(mnze(j),4),1:3);
```

```
0230            end
0231             for i=1:3
0232                ElementsToCheck((j-1)*6+i) = ...
0233                                ElementsOverEdge(Nodes(mod(i,3)+1),Nodes(i));
0234                if N4E(mnze(j),4)
0235                   ElementsToCheck((j-1)*6+3+i) = ...
0236                    ElementsOverEdge(FatherNodes(mod(i,3)+1),FatherNodes(i));
0237                end
0238             end
0239        end
0240        ElementsToCheck = setdiff(ElementsToCheck,0);
```

The additional nodes due to red-refinement are stored in $Grid.NOE$ (abbreviated by NOE). Since we know the oriented edges of $Grid$ we preallocate all the memory which is required by

```
0242        %prelocating memory for new nodes with arbitrary value -1
0243        I1 = [N4E(mnze,1); N4E(mnze,2); N4E(mnze,3)];
0244        I2 = [N4E(mnze,2); N4E(mnze,3); N4E(mnze,1)];
0245        NOE = NOE + sparse(I1,I2,-1,size(NOE,1),size(NOE,1));
0246        RealSizeC4N = size(C4N,1); %tracks the current size of C4N
0247        C4N = [C4N; ones(max(3*nnz(mnze)-nnz(Grid.Unused.C4N),0),2)];
0248        RealSizeN4E = size(N4E,1); %tracks the current size of C4N
0249        N4E = [N4E; ones(max(3*nnz(mnze)-nnz(Grid.Unused.N4E),0),5)];
0250        RealSizeF4E = size(F4E,1); %tracks the current size of C4N
0251        F4E = [F4E; ones(max(nnz(mnze)-nnz(Grid.Unused.F4E),0),8)];
```

In a loop over all marked elements and their edges, the algorithm now either stores an existing vertex in NOE, or creates a new one at the barycenter of the edge

```
0259            LocElement = N4E(mnze(j),1:4);
0260            FatherOfElement = LocElement(4);
0261            ElementEdges = [LocElement([1 2 3])',LocElement([2 3 1])'];
0262
0263            for n=1:3
0264                if NOE(ElementEdges(n,2),ElementEdges(n,1))>0
0265                %there is already a vertex on this edge
0266                    NewVertices(n) = ...
0267                                    NOE(ElementEdges(n,2),ElementEdges(n,1));
0268                    NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0269                              NOE(ElementEdges(n,2),ElementEdges(n,1));
0270                else
0271                    %create new vertex
0272                    if isempty(Grid.Unused.C4N)
0273                        NewVertices(n) = RealSizeC4N + 1;
0274                        NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0275                                                NewVertices(n); %update NOE
0276                        C4N(RealSizeC4N+1,:) = (C4N(ElementEdges(n,1),:) ...
0277                                              + C4N(ElementEdges(n,2),:))/2;
0278                        RealSizeC4N = RealSizeC4N + 1; %update size of C4N
0279                    else
0280                        C4N(Grid.Unused.C4N(1),:) = ...
```

```
0281                                          (C4N(ElementEdges(n,1),:) ...
0282                                           + C4N(ElementEdges(n,2),:))/2;
0283                    NewVertices(n) = Grid.Unused.C4N(1);
0284                    NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0285                                                    Grid.Unused.C4N(1);
0286                    Grid.Unused.C4N = ...
0287                              Grid.Unused.C4N(2:size(Grid.Unused.C4N,1));
0288                end
0289            end
```

The Dirichlet edges may have changed during this process. Thus, we update $Grid.N4D$ (abbreviated by $N4D$) by

```
0332      %Update Dirichlet edges
0333      NewDirichlet = [];
0334      for j=1:size(N4D,1)
0335         if NOE(N4D(j,1),N4D(j,2))
0336             NewDirichlet = [NewDirichlet; ...
0337                             N4D(j,1) NOE(N4D(j,1),N4D(j,2)); ...
0338                             NOE(N4D(j,1),N4D(j,2)) N4D(j,2)];
0339             NOE(N4D(j,1),N4D(j,2)) = 0;
0340             NOE(N4D(j,2),N4D(j,1)) = 0;
0341         else
0342             NewDirichlet =[NewDirichlet; N4D(j,:)];
0343         end
0344      end
0345      N4D = NewDirichlet;
```

and analogously for the Neumann edges $Grid.N4N$. Note, that we deleted the entries of NOE for the edges on the boundary, to identify nodes on the boundary in future coarsening processes. Before the red-refinement of the marked elements, we stored all neighboring triangles thereof. Thus, we now find the irregular elements of $Grid$ by simply counting the hanging nodes of these elements:

```
0361      %Because of new elements, grid condition may be violated
0362      nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0363      HangingNodes = abs(NOE-NOE');
0364
0365      if nnz(HangingNodes)
0366
0367          %Check only elements which have been identified via
0368          %ElementsToCheck
0369          for j=1:size(ElementsToCheck,2)
0370              LocElement = N4E(ElementsToCheck(j),1:3);
0371              ElementEdges = [LocElement([1 2 3])',LocElement([2 3 1])'];
0372              HangingEdgesCounter = 0;
0373              for n=1:3
0374                  if HangingNodes(ElementEdges(n,2),ElementEdges(n,1))
0375                     HangingNode = ...
0376                         HangingNodes(ElementEdges(n,2),ElementEdges(n,1));
0377                     if HangingNodes(ElementEdges(n,2),HangingNode) ...
```

```
0378                        || HangingNodes(HangingNode,ElementEdges(n,1))
0379                        MarkedElements = ...
0380                                    [MarkedElements; ElementsToCheck(j)];
0381                        break;
0382                     else
0383                        HangingEdgesCounter = HangingEdgesCounter + 1;
0384                     end
0385                  end
0386               end
0387               if HangingEdgesCounter>1
0388                  MarkedElements = [MarkedElements; ElementsToCheck(j)];
0389               end
0390            end
0391         end
```

After the *while*-loop, the structure *Grid* satisfies the grid condition, and a green completion gives a regular triangulation.

```
0414 function N4E = greenCompletion(N4E,NOE,UnusedElements)
0416
0417   nze = setdiff(1:size(N4E,1),UnusedElements);
0418   Nnze = length(nze);
0419   HangingNodes = abs(NOE-NOE');
0420   a = N4E(nze,1:3)';
0421   I1 = reshape(a,3*Nnze,1);
0422   a = N4E(nze,[2 3 1])';
0423   I2 = reshape(a,3*Nnze,1);
0424   a = repmat(1:Nnze,3,1);
0425   Val = reshape(a,3*Nnze,1);
0426   ElementsOnEdges = sparse(I1,I2,Val,size(NOE,1),size(NOE,2));
0427   [I1,I2] = find(HangingNodes);
0428   [I1,I2,ElementsToConsider] = ...
0429             find(sparse(I1,I2,1,size(NOE,1),size(NOE,2)).*ElementsOnEdges);
0430   ElementsToConsider = sort(nze(ElementsToConsider));
0431   %preallocation
0432   Counter = size(N4E,1);
0433   N4E = [N4E; zeros(nnz(HangingNodes)/2,5)];
0434
0436   for j=1:nnz(ElementsToConsider)
0437       Element = ElementsToConsider(j);
0438
0439       for k=1:3
0440            if HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k))
0441               HangingNode = ...
0442                       HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k));
0443               Counter = Counter + 1;
0444               LocElement = N4E(Element,:);
0445               N4E(Element,:) = ...
0446                      [HangingNode LocElement(rem(k+1,3)+1) LocElement(k)...
0447                                                  LocElement(4) Counter];
```

```
0448                N4E(Counter,:) = [HangingNode LocElement(rem(k,3)+1)...
0449                                  LocElement(rem(k+1,3)+1) LocElement(4) Element];
0450
0451                break;
0452            end
0453
0454        end
0455    end
0456
0457    N4E = N4E(1:Counter,:);
```

The complete code of *refineGrid* can be found in Appendix A.1.

1.3.2 The Function *coarsenGrid*

Similarly to the previous section, reversing a green completion gives a grid. Then, we find the marked last generation fathers in *Grid.F4E*.

```
0170    HigherGenerationFathers = unique(Grid.F4E(find(Grid.F4E(:,8)),8));
0171    LastGenerationFathers = setdiff(nzf,HigherGenerationFathers);
0172    Children = Grid.F4E(LastGenerationFathers,4:7);
0173    %marked last generation fathers, i.e the last generation fathers where
0174    %all children are marked
0175    mlgf = find(sum([MarkedElements(Children(:,1),2),...
0176                     MarkedElements(Children(:,2),2),...
0177                     MarkedElements(Children(:,3),2),...
0178                     MarkedElements(Children(:,4),2)],2)==4);
0179
0180    for j=1:size(mlgf,1)
0181          Grid = coarsenFather(LastGenerationFathers(mlgf(j)),Grid);
0182    end
```

Note, that the non-trivial entries in the second column of *MarkedElements* indicate the elements which are to be coarsened. However, an element os only coarsened provided that all children of its father are marked for coarsening. The coarsening of a specific father is carried out by the subfunction *coarsenFather*

```
0301 function Grid = coarsenFather(FatherToCoarse,Grid)
0302 %coarsenFather: red-coarsens FatherToCoarse
0303
0304        Children = Grid.F4E(FatherToCoarse,4:7);
0305        EdgesOfFather = [Grid.F4E(FatherToCoarse,[1 2 3])' ...
0306                         Grid.F4E(FatherToCoarse,[2 3 1])'];
0307
0308        for j=1:3
0309            if Grid.NOE(EdgesOfFather(j,2),EdgesOfFather(j,1))==0 ...
0310               && Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2))==0
0311                %BoundaryNode
0312                Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = -1;
0313            elseif (Grid.NOE(EdgesOfFather(j,2),EdgesOfFather(j,1))==0)
0314                %HangingNode
```

```
0315              Grid.C4N(Grid.NOE(EdgesOfFather(j,1),...
0316                       EdgesOfFather(j,2)),:) = [0 0];
0317              Grid.Unused.C4N =  ...
0318          [Grid.Unused.C4N; Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2))];
0319              Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = 0;
0320            else
0321              Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = 0;
0322            end
0323        end
0324
0325        Grid.N4E(Children(1),:) = [Grid.F4E(FatherToCoarse,1:3)...
0326                                   Grid.F4E(FatherToCoarse,8) 0];
0327        Grid.N4E(Children(2:4),:) = 0;
0328        Grid.Unused.N4E = [Grid.Unused.N4E; Children(2:4)'];
0329        Grid.F4E(FatherToCoarse,:) = 0;
0330        Grid.Unused.F4E = [Grid.Unused.F4E; FatherToCoarse];
```

Updating the Dirichlet and Neumann edges is done in a similar way as in the previous section. Finally, we satisfy the grid condition by simply calling $refineGrid$ in case of irregular elements. The complete code of *coarsenGrid* can be found in Appendix A.2.

Chapter 2

Finite Element Method

In this chapter, we specify the finite element method for the numerical solution of the parabolic problem (0.1). First, we define the necessary function spaces which will be used in the following sections. Then we discuss existence and uniqueness of the solution of the finite element discretization and its convergence in context of the elliptic problem. Finally, we focus on the parabolic problem and show that the time-space discretization thereof results in the characteristic matrix form of (0.1). Finally, we state the convergence properties of such a discretization.

2.1 Function Spaces

In the analysis, following function spaces and norms will be used. In all definitions let ω be a bounded open subset of Ω and γ its Lipschitz boundary. We will omit the indices and the set classifications if $\omega = \Omega$.

Definition 8. We denote by $L^2(\omega)$ the **Lebesgue Hilbert space** with the canonical scalar product

$$(v\ ;\ w)_\omega := \int_\omega v\, w\, dx$$

and the induced norm

$$\|v\|_\omega := \|v\|_{L^2(\omega)} = (v\ ;\ v)_\omega^{1/2}.$$

The analogous notation is used in case of $L^2(\gamma)$.

Furthermore, we define the **Sobolev spaces** $H^1(\omega)$ and $W^{1,\infty}(\omega)$ by

$$H^1(\omega) := \left\{v \in L^2(\omega) \text{ weakly differentiable} \,\middle|\, \nabla v \in L^2(\omega)^2\right\},$$
$$W^{1,\infty}(\omega) := \left\{v \in L^\infty(\omega) \text{ weakly differentiable} \,\middle|\, \nabla v \in L^\infty(\omega)^2\right\}.$$

By H^1_D we denote the subset of H^1 with functions that vanish on the Dirichlet boundary, i.e. $v := \left\{w \in H^1 \,\middle|\, w|_{\Gamma_D} = 0\right\}$. We equip H^1_D with a special norm, the natural energy norm of the problem (0.1).

Definition 9. For $v \in H^1(\omega)$ we denote by

$$|||v|||_\omega := \{\lambda\|\nabla v\|_\omega^2 + \beta\|v\|_\omega^2\}^{1/2} \tag{2.1}$$

the **energy norm** with λ and β defined in $(P2)$ and $(P3)$, respectively, see page 4.

The space H^1_D equipped with the energy norm $|||\cdot|||$ is a **Banach space**.

Definition 10. We denote by H^{-1} the **dual space** of H^1_D. We equip H^{-1} with the norm

$$|||\varphi|||_* := \sup_{v\in H^1_D\setminus\{0\}} \frac{\langle \varphi\,;\, v\rangle}{|||v|||}, \quad \text{for } \varphi \in H^{-1},$$

where the duality brackets $\langle \varphi\,;\, v\rangle := \langle \varphi\,;\, v\rangle_{H^{-1}\times H^1_D} = \varphi(v)$ extend the L^2-scalar product.

To measure functions on the boundary we introduce $H^{1/2}(\Gamma)$, the natural **space of traces** and equip it with the norm $\|w\|_{H^{(1/2)}(\Gamma)} := \inf\big\{|||v||| \,\big|\, v\in H^1, v|_\Gamma = w\big\}$. Analogously to Definition 10, we denote the corresponding dual space and its norm by

$$H^{-1/2}(\Gamma) := (H^{1/2}(\Gamma))^* \text{ and } |||\psi|||_* := \sup_{w\in H^{1/2}(\Gamma)\setminus\{0\}} \frac{\langle \psi\,;\, w\rangle}{\|w\|_{H^{1/2}(\Gamma))}} \quad \text{for } \psi\in H^{-1/2}(\Gamma),$$

where $\langle\cdot\,;\,\cdot\rangle$ denotes the extended L^2-scalar product of $L^2(\Gamma)$.

Definition 11. We define the **space-time Banach spaces** $L^2(a,b;V)$ and $L^\infty(a,b;V)$ by

$$L^2(a,b;V) := \{v : (a,b)\to V \text{ measurable} \mid t\mapsto \|v(.,t)\|_V \text{ is square integrable}\},$$
$$L^\infty(a,b;V) := \{v : (a,b)\to V \text{ measurable} \mid t\mapsto \|v(.,t)\|_V \text{ is essentially bounded}\},$$

and equip them with the norms

$$\|v\|_{L^2(a,b;V)} := \Big\{\int_a^b \|v(.,t)\|_V^2\, dt\Big\}^{1/2} \text{ and } \|v\|_{L^\infty(a,b;V)} := \operatorname*{ess\,sup}_{a<t<b} \|v(.,t)\|_V,$$

respectively. Finally, we introduce the **energy space** $X(a,b)$ for the parabolic problem:

Definition 12.

$$X(a,b) := \{v\in L^2(a,b;H^1_D)\cap L^\infty(a,b;L^2) : \partial_t v + \vec{c}(x,t)\cdot\nabla v \in L^2(a,b;H^{-1})\},$$
$$\|v\|_{X(a,b)} := \Big\{\|v\|^2_{L^\infty(a,b;L^2)} + \|v\|^2_{L^2(a,b;H^1_D)} + \int_a^b |||\partial_t v + \vec{c}(x,t)\cdot\nabla v|||_*^2\, dt\Big\}^{1/2}.$$

2.2 Elliptic Problem

Here, we consider the elliptic problem. First, we derive its so-called weak form, which leads to a variational problem for test functions $v\in H^1_D$. We also recall the unique solvability of the weak form. Finally, we derive the corresponding finite element discretization and show that the finite element approximations converge to the weak solution of the stationary problem.

2.2.1 Strong Form and Weak Form

We consider the elliptic model problem

$$-\operatorname{div}(D\nabla u) + \vec{c}\cdot\nabla u + ru = f \quad \text{in } \Omega, \tag{2.2a}$$

$$u = 0 \quad \text{on } \Gamma_D, \tag{2.2b}$$

$$\vec{n}\cdot D\nabla u = g \quad \text{on } \Gamma_N, \tag{2.2c}$$

with solution $u \in \mathbb{R}$. The functions $D \in \mathbb{R}^{2\times 2}$, $\vec{c} \in \mathbb{R}^2$, $r \in \mathbb{R}$, $f \in \mathbb{R}$ are given function of x in Ω and $g \in \mathbb{R}$ is a function of x on Γ_N. Equation (2.2) is the so-called **strong form** of the problem. To obtain the weak form of the problem we multiply the differential equation (2.2a) by a test function $v \in H^1_D$ and integrate over Ω. Thus, we obtain

$$-\int_\Omega \operatorname{div}(D\nabla u)\, v\, dx + \int_\Omega \vec{c}\cdot\nabla u\, v\, dx + \int_\Omega r\, u\, v\, dx = \int_\Omega f\, v\, dx. \tag{2.3}$$

We use integration by parts to rewrite the first term in the above equation,

$$-\int_\Omega \operatorname{div}(D\nabla u)\, v\, dx = -\int_\Gamma \vec{n}\cdot(D\nabla u)\, v\, ds + \int_\Omega (D\nabla u)\cdot\nabla v\, dx.$$

Since v vanishes on the Dirichlet boundary Γ_D, we can replace Γ by Γ_N in the previous equation and therefore,

$$-\int_\Omega \operatorname{div}(D\nabla u)\, v\, dx = -\int_{\Gamma_N} \underbrace{\vec{n}\cdot(D\nabla u)}_{=g}\, v\, ds + \int_\Omega (D\nabla u)\cdot\nabla v\, dx. \tag{2.4}$$

With (2.4) we obtain the following **weak form** of (2.2): Find u such that

$$\begin{aligned}(\nabla v\,;\, D\nabla u) + (v\,;\,\vec{c}\cdot\nabla u) + (v\,;\,ru) = (v\,;\,f) + (v\,;\,g)_{\Gamma_N} \quad \forall v \in H^1_D,\\ u = 0 \quad \text{on } \Gamma_D.\end{aligned} \tag{2.5}$$

Provided $D \in (L^\infty)^{2\times 2}$, $\vec{c} \in (W^{1,\infty})^2$, $r \in L^\infty$, $f \in L^2$ and $g \in L^2(\Gamma_N)$, all occurring L^2-scalar products exist according to the Hölder inequality.

Remark.

(i) From the derivation of the weak form (2.5) it is clear that each solution $u \in C^2(\overline{\Omega})$ of the strong form (2.2) is necessarily a solution of the weak form.

(ii) If $u \in H^1_D$ is a solution of (2.5) and $D\nabla u \in H(\operatorname{div})$, an integration by parts yields

$$(v\,;\,-\operatorname{div}(D\nabla u) + \vec{c}\cdot\nabla u + ru - f) = 0 \quad \forall v \in \mathcal{D}(\Omega) \subseteq H^1_D.$$

Therefore, it follows from the fundamental theorem of the calculus of variations, c.f. Praetorius [13, p.21], that u solves (2.2a) almost everywhere.

(iii) In the following, we will use the Lax-Milgram lemma, c.f. Evans [11, §6.2,p.297], to prove that the weak form (2.5) has a unique solution $u \in H^1$.

2.2.2 Existence and Uniqueness

We now apply the lemma of Lax-Milgram to show the existence and uniqueness of a solution u of the weak form (2.5). For the proof, ellipticity and continuity is required on H^1_D. These two conditions mean certain restrictions on the problem data and therefore, we need to make the following assumptions:

ASSUMPTION.

(E1) The coefficient functions satisfy $D \in (L^\infty)^{2\times 2}$, $\vec{c} \in (W^{1,\infty})^2$, $r \in L^\infty$. Moreover, we assume $f \in L^2$ and $g \in L^2(\Gamma_N)$.

(E2) $D \in \mathbb{R}^{2\times 2}$ is symmetric and uniformly positive definite on Ω, i.e. with

$$\lambda(x) := \min_{z\in\mathbb{R}^2\setminus\{0\}} \frac{z \cdot D(x)z}{|z|^2} > 0 \tag{2.6}$$

the constant

$$\lambda_0 := \operatorname*{ess\,inf}_{x\in\Omega} \lambda(x) \tag{2.7}$$

is positive. Furthermore, if we define

$$\kappa(x) :=\lambda_0^{-1} \max_{z\in\mathbb{R}^2\setminus\{0\}} \frac{z \cdot D(x)z}{|z|^2} \quad . \tag{2.8}$$

there holds

$$\kappa_0 := \operatorname*{ess\,sup}_{x\in\Omega} \kappa(x) < \infty. \tag{2.9}$$

Note that $\kappa_0 \geq 1$.

(E3) There exists a constant β_0, such that

$$\beta_0 := \operatorname*{ess\,inf}_{x\in\Omega} \left\{ r - \frac{1}{2} \operatorname{div} \vec{c} \right\} \geq 0. \tag{2.10}$$

(E4) There exists a constant $c_{r,0} \geq 0$, such that

$$\|r\|_{L^\infty} \leq c_{r,0}\beta_0, \tag{2.11}$$

with β_0 from (2.10).

(E5) On the Neumann boundary there is only outflow convection, i.e. for all $x \in \Gamma_N$, $\vec{c}(x) \cdot \vec{n}(x) \geq 0$ holds almost everywhere.

Remark. The energy norm of the elliptic problem is given in (2.1), with constants $\lambda = \lambda_0$ and $\beta = \beta_0$.

The constant λ_0 from Assumption $(E2)$ is a lower bound for the smallest eigenvalue of D in Ω. On the other hand, the constant κ_0 measures the ratio of the largest and the smallest eigenvalue of D and therefore the stiffness of the problem. As stated in Section 2.2, Assumption $(E1)$ ensures the existence of the occurring scalar products and more generally, the weak form (2.5). The relevance of the other assumptions becomes clear in view of the following two lemmata:

Lemma 2.1 (Ellipticity). *Assumptions* $(E1)$, $(E2)$, $(E3)$ *and* $(E5)$ *imply*

$$(\nabla v\,;\,D\nabla v) + (v\,;\,\vec{c}\cdot\nabla v) + (v\,;\,rv) \geq |\!|\!| v |\!|\!|^2 \quad \forall\, v \in H^1_D. \tag{2.12}$$

Proof. Let $v \in H^1_D$ and let us recall the definition of the energy norm

$$|\!|\!| v |\!|\!|^2 \overset{(2.1)}{=} \lambda_0 \|\nabla v\|^2 + \beta_0 \|v\|^2 = \lambda_0 \int_\Omega |\nabla v|^2\,dx + \beta_0 \int_\Omega v^2\,dx.$$

Now, we estimate both terms on the right hand side. For the first one, we use (2.6) and (2.7),

$$\lambda_0 \int_\Omega |\nabla v|^2\,dx \overset{(2.7)}{\leq} \int_\Omega (\lambda(x)\nabla v)\cdot(\nabla v)\,dx \overset{(2.6)}{\leq} \int_\Omega (D\nabla v)\cdot(\nabla v)\,dx = (\nabla v\,;\,D\nabla v). \tag{2.13}$$

The bound for the second one follows from $r - \frac{1}{2}\operatorname{div}\vec{c} \geq \beta_0$, c.f. Assumption $(E3)$,

$$\beta_0 \int_\Omega v^2\,dx \overset{(2.10)}{\leq} \int_\Omega \Big\{ r - \frac{1}{2}\operatorname{div}\vec{c} \Big\} v^2\,dx = (v\,;\,rv) - \frac{1}{2}\int_\Omega (\operatorname{div}\vec{c})\; v^2\,dx. \tag{2.14}$$

Using the product rule $(\operatorname{div}\vec{c})\; v^2 = \operatorname{div}(\vec{c}\; v^2) - 2(\vec{c}\cdot\nabla v)v$ and integration by parts, we can rewrite the last term in (2.14) and obtain

$$\begin{aligned}
-\frac{1}{2}\int_\Omega \operatorname{div}\vec{c}\, v^2\,dx &= -\frac{1}{2}\int_\Gamma (\vec{n}\cdot\vec{c}) \underbrace{v^2}_{=0 \text{ on } \Gamma_D}\,ds + \int_\Omega (\vec{c}\cdot\nabla v)v\,dx \\
&= -\frac{1}{2}\int_{\Gamma_N} \underbrace{(\vec{n}\cdot\vec{c})}_{\substack{(E5)\\ \geq 0}} v^2\,ds + \int_\Omega (\vec{c}\cdot\nabla v)v dx. \\
&\leq (v\,;\,\vec{c}\cdot\nabla v).
\end{aligned}$$

Now the result follows from the above inequality together with (2.13) and (2.14). ∎

Lemma 2.2 (Continuity). *Assumptions* $(E1)$, $(E2)$, *and* $(E4)$ *imply*

$$(\nabla v\,;\,D\nabla w) + (v\,;\,\vec{c}\cdot\nabla w) + (v\,;\,rw) \lesssim |\!|\!| v |\!|\!|\; |\!|\!| w |\!|\!| \quad \forall v, w \in H^1_D, \tag{2.15}$$

with a constant that depends on λ_0, κ_0 *from* $(E2)$, β_0 *from* $(E3)$, $c_{r,0}$ *from* $(E4)$, $\|\vec{c}\|_{L^\infty}$, *the shape, and the diameter of* Ω.

Proof. Let $w, v \in H^1_D$. We estimate the left hand side of (2.15) using the Cauchy's inequality,

$$(\nabla v\,;\,D\nabla w) + (v\,;\,\vec{c}\cdot\nabla w) + (v\,;\,rw) \overset{\text{Cauchy}}{\leq} \|\nabla v\|\|D\nabla w\| + \|v\|\|\vec{c}\cdot\nabla w\| + \|v\|\|rw\|.$$

The term $\|D\nabla w\| = \Big\{ \int_\Omega |D(x)\nabla w|^2\,dx \Big\}^{1/2}$ is estimated using $(E2)$. Furthermore, Assumption $(E4)$ and $c_{\vec{c},0} := \|\vec{c}\|_{L^\infty}$ imply

$$\begin{aligned}
&(\nabla v\,;\,D\nabla w) + (v\,;\,\vec{c}\cdot\nabla w) + (v\,;\,rw) \\
&\qquad \leq \underbrace{\operatorname*{ess\,sup}_{x\in\Omega} \max_{z\in\mathbb{R}^2\setminus\{0\}} \frac{z\cdot D(x)z}{|z|^2}}_{\overset{(2.9)}{=}\kappa_0\lambda_0} \|\nabla v\|\|\nabla w\| + c_{\vec{c},0}\|v\|\|\nabla w\| + c_{r,0}\beta_0\|v\|\|w\|.
\end{aligned}$$

Note that $\kappa_0 \geq 1$ and $\max\{c_{r,0}, 1\} \geq 1$, so that

$$\begin{aligned}(\nabla v\,;\, D\nabla w) + (v\,;\, \vec{c}\cdot\nabla w) + (v\,;\, rw)\\ \leq \kappa_0 \max\{c_r, 1\}(\lambda_0\|\nabla w\|\,\|\nabla v\| + \beta_0\|v\|\,\|w\|) + c_{\vec{c},0}\|v\|\,\|\nabla w\|.\end{aligned}$$

From the Friedrichs inequality[1] we have

$$\|v\| \leq \min\{c_P \operatorname{diam}(\Omega)\lambda_0^{-1/2}, \beta_0^{-1/2}\}|\!|\!|v|\!|\!|$$

with a constant c_P that only depends on the shape of Ω. Moreover,

$$\|\nabla w\| = \lambda_0^{-1/2}\{\lambda_0\|\nabla w\|^2\}^{1/2} \leq \lambda_0^{-1/2}|\!|\!|w|\!|\!|.$$

Cauchy's inequality in $\mathbb{R}^2$ yields[2]

$$\begin{aligned}&(\nabla v\,;\, D\nabla w) + (v\,;\, \vec{c}\cdot\nabla v) + (v\,;\, rw)\\ &\quad\lesssim \left\{\lambda_0\|\nabla w\|^2 + \beta_0\|w\|^2\right\}^{1/2}\left\{\lambda_0\|\nabla v\|^2 + \beta_0\|v\|^2\right\}^{1/2} + \lambda_0^{-1/2}|\!|\!|v|\!|\!|\,|\!|\!|w|\!|\!|\\ &\quad\lesssim |\!|\!|v|\!|\!|\,|\!|\!|w|\!|\!|\end{aligned}$$

and this completes the proof. ■

Thus, if the problem data satisfies Assumptions (E1)–(E5), a unique solution of the weak form (2.5) exists. Moreover, without the convection term on the left hand side of (2.15), we can identify the constants more precisely:

Corollary 2.3. *Under the assumptions of Lemma 2.2, there holds*

$$(\nabla v\,;\, D\nabla w) + (v\,;\, rw) \leq \kappa_0 \max\{c_r, 1\}|\!|\!|v|\!|\!|\,|\!|\!|w|\!|\!| \quad \forall v, w \in H^1_D. \tag{2.17}$$

Proof. The result follows by repeating the proof of (2.15) without $(v\ ;\vec{c}\cdot\nabla w)$ on the left hand side. ■

2.2.3 Finite Element Discretization

Essentially, to apply the finite element method means to discretize the space H^1_D by a finite dimensional subspace. The Céa-lemma[3] states that the quality of the approximation vitally depends on this subspace. For simplicity, we choose a finite element space consisting of linear basis functions:

Definition 13. Let $\mathcal{T}$ be a regular triangulation and $\mathcal{N}_f$ the set of all free nodes of $\mathcal{T}$. Then, we define by $\mathcal{S}^1_D(\mathcal{T})$ the corresponding P_1 **finite element space**. The space $\mathcal{S}^1_D(\mathcal{T})$ is a finite dimensional subspace of H^1_D and consists of continuous functions which are piecewise affine on all elements of $\mathcal{T}$. We consider the so-called **nodal basis** $\{\phi_1, \ldots, \phi_m\}$ of $\mathcal{S}^1_D(\mathcal{T})$, which consists of $m = |\mathcal{N}_f|$ globally continuous and $\mathcal{T}$-piecewise affine functions ϕ_i that satisfy

[1] see Section 3.1.

[2]
$$ab + cd = \begin{pmatrix}a\\c\end{pmatrix}\cdot\begin{pmatrix}b\\d\end{pmatrix} \overset{\text{Cauchy}}{\leq} \left\|\begin{pmatrix}a\\c\end{pmatrix}\right\|\left\|\begin{pmatrix}b\\d\end{pmatrix}\right\| \leq (a^2+c^2)^{1/2}(b^2+d^2)^{1/2} \tag{2.16}$$

[3] see Braess [5, p. 53]

$\phi_i(x_j) = \delta_{ij}$, where $x_j \in \mathcal{N}_f$ is the j^{th} free node.

In P_1-FEM, one tries to find a solution from $\mathcal{S}^1_D(\mathcal{T})$ which satisfies the weak formulation (2.5). This means that one wants to find $u_h \in \mathcal{S}^1_D(\mathcal{T})$ such that

$$(\nabla v_h \; ; D\nabla u_h) + (v_h \; ; \vec{c} \cdot \nabla u_h) + (v_h \; ; r u_h) = (v_h \; ; f) \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}). \tag{2.18}$$

Since (2.18) is linear with respect to v_h, it immediately follows that (2.18) is equivalent to the system of m linear equations

$$(\nabla \phi_i \; ; D\nabla u_h) + (\phi_i \; ; \vec{c} \cdot \nabla u_h) + (\phi_i \; ; r u_h) = (\phi_i \; ; f) \text{ with } i = 1, \ldots, m. \tag{2.19}$$

If we now replace u_h by its basis representation $\sum_{j=1}^m u_{h,j}\phi_j$ with coefficients $u_{h,j} \in \mathbb{R}$ we obtain

$$\sum_{j=1}^m (\nabla \phi_i \; ; D\nabla \phi_j) u_{h,j} + \sum_{j=1}^m (\phi_i \; ; \vec{c} \cdot \nabla \phi_j) u_{h,j} + \sum_{j=1}^m (\phi_i \; ; r\phi_j) u_{h,j} = (\phi_i \; ; f) \text{ for } i = 1, \ldots, m. \tag{2.20}$$

The matrix $\underline{D} \in \mathbb{R}^{m \times m}$ with $\underline{D}_{ij} := (\nabla \phi_i \; ; D\nabla \phi_j)$ is called the **stiffness matrix**, $\underline{C} \in \mathbb{R}^{m \times m}$ with $\underline{C}_{ij} := (\phi_i \; ; \vec{c} \cdot \nabla \phi_j)$ is called the **convection matrix** and $\underline{R} \in \mathbb{R}^{m \times m}$ with $\underline{R}_{ij} := (\phi_i \; ; r\phi_j)$ is called the **reaction matrix**. Furthermore, we define vectors $\vec{u}$, $\vec{b} \in \mathbb{R}^m$ by $\vec{u}_j := u_{h,j}$ and $\vec{b}_i := (\phi_i \; ; f)$, respectively. We then obtain the matrix equation, which is equivalent to (2.18)

$$(\underline{D} + \underline{C} + \underline{R})\vec{u} = \vec{b}. \tag{2.21}$$

The regularity of the matrix on the left hand side is equivalent to the existence of a unique solution $\vec{u}$ of (2.21) for any right hand side $\vec{b}$. Since $\mathcal{S}^1_D(\mathcal{T})$ is a closed subspace of H^1_D, it follows from the ellipticity (2.12), the continuity (2.15) and the Lax-Milgram lemma, that (2.21) has a unique solution. Thus, the matrix $(\underline{D} + \underline{C} + \underline{R})$ is regular.

2.2.4 Convergence

Convergence is a crucial property for any numerical method and it means that the approximations u_h obtained from the FEM discretization approach u when $h_{max} := \max\limits_{K \in \mathcal{T}} h_K$ tends to zero. In the following lemma convergence of the FEM scheme with respect to the energy norm (2.1) for the functions D, $\vec{c}$ and r from (2.2) is discussed. Let us first define an operator which is a projection on the finite element space $\mathcal{S}^1_D(\mathcal{T})$:

Definition 14. Let $a(v,u) := (\nabla v \; ; D\nabla u) + (v \; ; \vec{c} \cdot \nabla u) + (v \; ; ru)$ be an elliptic and continuous bilinear form, i.e. for $v, u \in H^1_D$ Equation (2.12) and (2.15) hold. Then, according to the Lax-Milgram lemma, there exists an unique $u_h \in \mathcal{S}^1_D(\mathcal{T})$, such that

$$a(v_h, u_h) = a(v_h, u) \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}). \tag{2.22}$$

Thus, the operator $G_h : H^1_D \mapsto \mathcal{S}^1_D(\mathcal{T})$, defined by $G_h u := u_h$, is well defined. The linear operator G_h is denoted by **Galerkin operator** and the equation (2.22) is called **Galerkin orthogonality**.

Lemma 2.4. *Let $u \in H^1_D$ be the weak solution of (2.3). Let $\left(\mathcal{T}_n\right)_{n \in \mathbb{N}}$ be a sequence of regular triangulations of Ω, such that $\sup\limits_{n \in \mathbb{N}} \sup\limits_{K \in \mathcal{T}_n} h_K^2/|K| \leq \infty$ and $h_{max,n} := \max\limits_{K \in \mathcal{T}_n} h_K \to 0$. For $n \in \mathbb{N}$ let $u_n \in \mathcal{S}^1_D(\mathcal{T}_n)$ be the Galerkin solution $G_{h,n}u = u_n$. Then, $\lim\limits_{n \to \infty} u_n = u$ in H^1_D.*

Proof. Let $u \in H^1_D$, $n \in \mathbb{N}$ with $G_{h,n}u := G_h u = u_n$, where G_h is the Galerkin operator of the n^{th} triangulation $\mathcal{T}_n$.
1. Claim. *G_h is bounded,*

$$\|\!|G_h|\!\| \leq c_1, \tag{2.23}$$

where c_1 is the constant from the continuity (2.15)*.*

Let $v \in H^1_D$ and recall the Galerkin orthogonality (2.22),

$$(\nabla w_h \,;\, D\nabla(v - G_h v)) + (w_h \,;\, \vec{c}\cdot\nabla(v - G_h v)) + (w \,;\, r(v - G_h v)) = 0 \quad \forall w_h \in \mathcal{S}^1_D(\mathcal{T}_n).$$

Thus, with the ellipticity (2.12) and the continuity (2.15) we obtain

$$\begin{aligned}\|\!|G_h v|\!\|^2 &\leq (D\nabla G_h v \,;\, \nabla G_h v) + (\vec{c}\cdot\nabla G_h v \,;\, G_h v) + (rG_h v \,;\, G_h v)\\ &= (D\nabla v \,;\, \nabla G_h v) + (\vec{c}\cdot\nabla v \,;\, G_h v) + (rv \,;\, G_h v)\\ &\lesssim \|\!|v|\!\|\,\|\!|G_h v|\!\|.\end{aligned}$$

We divide both sides by $\|\!|v|\!\|\,\|\!|G_h v|\!\|$ and (2.23) follows. □

2. Claim. *For any $\varepsilon > 0$ there exists $v \in \mathcal{C}^\infty_D(\overline{\Omega})$ such that*

$$\|\!|u - v|\!\| \leq \varepsilon. \tag{2.24}$$

Moreover, for $v \in \mathcal{C}^\infty_D(\overline{\Omega})$,

$$\|\!|v - G_h v|\!\| \leq c_2 \|\!|D^2 v|\!\| h_{max,n}, \tag{2.25}$$

where c_2 is a constant which only depends on λ_0 from $(E2)$, β_0 from $(E3)$, the ratios $h_K^2/|K|$ with $K \in \mathcal{T}_n$, and Ω.

The first equation follows from the density of $\mathcal{C}^\infty_D(\overline{\Omega})$ in H^1_D. The second inequality is a central FEM proposition, the so-called approximation theorem, c.f. Braess [5, Chapter II, §7, Th. 7.3, p.86] . Note that the constant depends on λ_0 from $(E2)$, on β_0 from $(E3)$, on the ratios $h_K^2/|K|$ with $K \in \mathcal{T}_n$, and on Ω, but not on v or $h_{max,n}$. □

Now let $v \in \mathcal{C}^\infty_D(\overline{\Omega})$. With (2.23) and (2.25) we have

$$\begin{aligned}\|\!|u - u_h|\!\| &\leq \|\!|u - v|\!\| + \|\!|v - G_h v|\!\| + \|\!|G_h v - G_h u|\!\|\\ &\leq \|\!|u - v|\!\| + c_2\|\!|D^2 v|\!\| h_{max,n} + \|\!|G_h|\!\|\,\|\!|u - v|\!\|\\ &\leq \|\!|u - v|\!\|(1 + c_1) + c_2\|\!|D^2 v|\!\| h_{max,n}.\end{aligned}$$

For a given $\varepsilon > 0$ we now choose v such that

$$\|\!|u - v|\!\| \leq \frac{\varepsilon}{2(1 + c_1)}$$

and $h_{max,n}$ such that

$$c_2\|\!|D^2 v|\!\| h_{max,n} \leq \frac{\varepsilon}{2}.$$

Consequently, the convergence result follows. ■

2.3 Parabolic Problem

After these considerations of the elliptic equation, we return to the parabolic problem (0.1). We now have to take into account that all functions in the equation depend on time.

2.3.1 Strong Form and Weak Form

Recall, the **strong form** of the problem (0.1) from page 4,

$$\begin{aligned} \frac{\partial u}{\partial t} - \operatorname{div}(D\nabla u) + \vec{c}\cdot\nabla u + ru &= f \quad && \text{in } \Omega\times(0,T],\\ u &= 0 && \text{on } \Gamma_D\times(0,T],\\ \vec{n}\cdot D\nabla u &= g && \text{on } \Gamma_N\times(0,T],\\ u &= u_0 && \text{in } \Omega \text{ for } t=0. \end{aligned}$$

The corresponding **weak form** for this problem can be derived in the same way as for the elliptic problem,

$$(v\,;\partial_t u)+(\nabla v\,;D\nabla u)+(v\,;\vec{c}\cdot\nabla u)+(v\,;ru)=(v\,;f)+(v\,;g)_{\Gamma_N}\quad \forall v\in H^1_D, t\in(0,T]. \tag{2.26}$$

2.3.2 Existence and Uniqueness of Solutions to the Parabolic Problem

Again, if a solution u of the weak form (2.26) exists and u is sufficiently smooth it follows from the fundamental theorem of the calculus of variations, c.f. Praetorius [13, p.21], that u also solves (0.1). We now follow the arguments from Evans [11, §7]. If the solution u exists one can show, that the time derivative $\partial_t u(\cdot,t)\in H^{-1}$ for $0<t\le T$ a.e., c.f. Evans [11, §7, p.352]. If we furthermore assume that $\partial_t u\in L^2(0,T;H^{-1})$ and $u\in L^2(0,T;H^1_0)$, then one can show that $u\in\mathcal{C}(0,T;L^2)$, where $\mathcal{C}(0,T;L^2)$ consists of all continuous functions $v:[0,T]\to L^2$ with $\|v\|_{\mathcal{C}(0,T;L^2)}:=\max\limits_{0\le t\le T}\|v(\cdot,t)\|<\infty$, c.f. Evans [11, §5, Theorem 3, p.287].

Analogously to Section 2.2.2, we assume $(P1)$–$(P5)$ to guarantee that all scalar products of the weak form exist and that the bilinear form of $(0.1)_a$ is elliptic and continuous in space for $0<t\le T$. Thus, the Lemmata 2.1 and 2.2 hold pointwise in the time interval $(0,T]$. Furthermore, comparing $(P1)$ with $(E1)$ shows, that the data functions D, $\vec{c}$ and r have to satisfy the ellipticity condition $(E1)$ for $0<t\le T$ and have to be continuous with respect to time. The other data functions have to satisfy this condition only almost everywhere.

Theorem 2.5 (Existence and Uniqueness, Evans [11, §2, Theorem 3 and 4, p.356–358]). *Assumptions $(P1)$–$(P5)$ imply, that there exists a unique weak solution of (0.1).* ■

2.3.3 Time Discretization and Spatial Discretization

For the time-discretization we choose an integer $N\ge 1$ and time steps $0=t_0<t_1<\ldots<t_N=T$ with local step sizes $\tau_n:=t_n-t_{n-1}$, where $1\le n\le N$. Throughout this section, we fix an arbitrary time step t_n and the corresponding time interval $(t_{n-1},t_n]$. For this time step t_n we denote the discrete solution and the data functions always with a superscript, e.g. $u_h^{(n)}(x):=u_h(x,t_n)$, $D^{(k)}(x):=D(x,t_n)$, $\vec{c}^{(n)}(x):=\vec{c}(x,t_n)$ and $r^{(n)}(x):=\vec{c}(x,t_n)$. Analogously, we deal with all other time dependent entities, such as for example the triangulation $\mathcal{T}^{(n)}$ and the finite element space $\mathcal{S}^1_D(\mathcal{T}^{(n)})$. The time derivative itself is discretized on the given interval by the finite difference scheme,

$$\frac{\partial u}{\partial t}\approx\frac{u_h^{(n)}-u_h^{(n-1)}}{\tau_n}.$$

Since the data functions are time dependent, we have to decide at which of the two time levels they will be evaluated. Here, we consider the so-called **θ-scheme**[4] for a fixed time discretization parameter $\theta \in [\frac{1}{2}, 1]$. We thus consider a convex combination of backward and forward Euler scheme of the equation (0.1). For $\theta = 1/2$ the method is called **Crank-Nicholson-scheme**, whereas if $\theta = 1$ it becomes the backward Euler scheme. The spatial discretization is obtained by replacing $u^{(n)}, u^{(n-1)} \in H^1_D$ by $u_h^{(n)} \in \mathcal{S}^1_D(\mathcal{T}^{(n)})$ and $u_h^{(n-1)} \in \mathcal{S}^1_D(\mathcal{T}^{(n-1)})$, respectively. Moreover, the test functions $v \in H^1_D$ of the weak form (2.26) are replaced by $v_h \in \mathcal{S}^1_D(\mathcal{T}^{(n)})$. The **backward Euler scheme** of the weak form of (0.1) reads

$$\begin{aligned}\frac{1}{\tau_n}(v_h\,;\,u_h^{(n)} - u_h^{(n-1)}) + (\nabla v_h\,;\,D^{(n)}\nabla u_h^{(n)}) + (v_h\,;\,\vec{c}^{(n)}\cdot\nabla u_h^{(n)}) + (v_h\,;\,r^{(n)}u_h^{(n)}) \\ = (v_h\,;\,f^{(n)}) + (v_h\,;\,g^{(n)})_{\Gamma_N} \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}^{(n)}),\end{aligned} \tag{2.27}$$

while the **forward Euler scheme** has the form

$$\begin{aligned}\frac{1}{\tau_n}(v_h\,;\,u_h^{(n)} - u_h^{(n-1)}) + (\nabla v_h\,;\,D^{(n-1)}\nabla u_h^{(n-1)}) + (v_h\,;\,\vec{c}^{(n-1)}\cdot\nabla u_h^{(n-1)}) + (v_h\,;\,r^{(n-1)}u_h^{(n-1)}) \\ = (v_h\,;\,f^{(n-1)}) + (v_h\,;\,g^{(n-1)})_{\Gamma_N} \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}^{(n)}).\end{aligned} \tag{2.28}$$

For both schemes the initial condition reads

$$u_h^{(0)} = \Pi_0 u_0, \tag{2.29}$$

where Π_0 is the L^2-projection onto $\mathcal{S}^1_D(\mathcal{T}^{(0)})$. For the θ-scheme we combine the equations (2.27) and (2.28),

$$\begin{aligned}\frac{1}{\tau_n}(v_h\,;\,u_h^{(n)} - u_h^{(n-1)}) + (\nabla v_h\,;\,\theta D^{(n)}\nabla u_h^{(n)} + (1-\theta)D^{(n-1)}\nabla u_h^{(n-1)}) \\ + (v_h\,;\,\theta\vec{c}^{(n)}\cdot\nabla u_h^{(n)} + (1-\theta)\vec{c}^{(n-1)}\cdot\nabla u_h^{(n-1)}) \\ + (v_h\,;\,\theta r^{(n)}u_h^{(n)} + (1-\theta)r^{(n-1)}u_h^{(n-1)}) \\ = (v_h\,;\,f^{(n),\theta}) + (v_h\,;\,g^{(n),\theta})_{\Gamma_N} \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}^{(n)})\end{aligned} \tag{2.30}$$

with $f^{(n),\theta} := \theta f^{(n)} + (1-\theta)f^{(n-1)}$ and $g^{(n),\theta} := \theta g^{(n)} + (1-\theta)g^{(n-1)}$. Unfortunately, it turns out that the above space-time discretization is not feasible for the a posteriori error analysis, since we would like the error estimator to consist of separate contributions of the data functions and of the discrete solution. Therefore, we introduce the following discretization:

$$\begin{aligned}\frac{1}{\tau_n}(v_h\,;\,u_h^{(n)} - u_h^{(n-1)}) + (\nabla v_h\,;\,D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})) \\ + (v_h\,;\,\vec{c}^{(n),\theta}\cdot(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})) \\ + (v_h\,;\,r^{(n),\theta}(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})) \\ = (v_h\,;\,f^{(n),\theta}) + (v_h\,;\,g^{(n),\theta})_{\Gamma_N} \quad \forall v_h \in \mathcal{S}^1_D(\mathcal{T}^{(n)})\end{aligned} \tag{2.31}$$

with $D^{(n),\theta} := \theta D^{(n)} + (1-\theta)D^{(n-1)}$, $\vec{c}^{(n),\theta} := \theta\vec{c}^{(n)} + (1-\theta)\vec{c}^{(n-1)}$ and $r^{(n),\theta} := \theta r^{(n)} + (1-\theta)r^{(n-1)}$ being the θ-weighted time discretizations of the coefficient functions. Again, we use the discretized initial condition (2.29).

[4]The reason for this approach are stability considerations, c.f. Verfürth [17, §3, Lemma III.4.4, p.126].

Remark. We did not derive this equation from the weak form (2.26), but, it will become clear from the a posteriori error analysis, that there exists a reliable and efficient error estimator for the error of the solution of (2.31).

With the nodal basis functions $\{\phi_1^{(n)}, \dots, \phi_m^{(n)}\}$ of $\mathcal{S}_D^1(\mathcal{T}^{(n)})$, Equation (2.31) becomes a system of equations, as it was the case in the previous section. Now the time discretizations $D^{(n),\theta}$, $\vec{c}^{(n),\theta}$, $r^{(n),\theta}$ replace their counterparts from the elliptic equation in the definition of the matrices, i.e. $\underline{D}^{(n),\theta}$, $\underline{C}^{(n),\theta}$, $\underline{R}^{(n),\theta} \in \mathbb{R}^{m\times m}$ with $\underline{D}_{ij}^{(n),\theta} := (\nabla\phi_i^{(n)} \,;\, D^{(n),\theta}\nabla\phi_j^{(n)})$, $\underline{C}_{ij}^{(n),\theta} := (\phi_i^{(n)} \,;\, \vec{c}^{(n),\theta}\cdot\nabla\phi_j^{(n)})$, $\underline{R}_{ij}^{(n),\theta} := (\phi_i^{(n)} \,;\, r^{(n),\theta}\phi_j^{(n)})$, where $m := |\mathcal{N}_f^{(n)}|$ is the number of free nodes at the n^{th} time step. Note that there are two major differences: First of all, the time derivative provides an additional term, the **mass matrix** $\underline{M}^{(n)} \in \mathbb{R}^{m\times m}$ defined by $\underline{M}_{ij}^{(n)} := (\phi_i^{(n)} \,;\, \phi_j^{(n)})$. Moreover, in the entities

$$(v_h \,;\, u_h^{(n-1)}),\ (\nabla v_h \,;\, D^{(n),\theta}\nabla u_h^{(n-1)}),\ (v_h \,;\, \vec{c}^{(n),\theta}\cdot\nabla u_h^{(n-1)}),\ (v_h \,;\, r^{(n),\theta}u_h^{(n-1)}),$$

v_h and $u_h^{(n-1)}$ are defined on two different triangulations $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$. If $\{\phi_1^{(n-1)}, \dots, \phi_{m'}^{(n-1)}\}$ is the nodal basis of $\mathcal{S}_D^1(\mathcal{T}^{(n-1)})$, we thus need to introduce the **transfer matrix** $\underline{T}^{(n),\theta} \in \mathbb{R}^{m\times m'}$ by

$$\begin{aligned}\underline{T}_{ij}^{(n),\theta} := (\phi_i^{(n)} \,;\, \phi_j^{(n-1)}) - \tau_n(1-\theta)\Big\{&(\nabla\phi_i^{(n)} \,;\, D^{(n),\theta}\nabla\phi_j^{(n-1)}) \\ &+ (\phi_i^{(n)} \,;\, \vec{c}^{(n),\theta}\cdot\nabla\phi_j^{(n-1)}) + (\phi_i^{(n)} \,;\, r^{(n),\theta}\phi_j^{(n-1)})\Big\}.\end{aligned}$$

With these settings, Equation (2.31) can be rewritten as the following system for the parabolic problem (0.1):

$$\Big\{\underline{M}^{(n),\theta} + \tau_n\theta(\underline{D}^{(n),\theta} + \underline{C}^{(n),\theta} + \underline{R}^{(n),\theta})\Big\}\vec{u}^{(n)} = \vec{b}^{(n),\theta}, \tag{2.32}$$

where $\vec{u}^{(n)}, \vec{b}^{(n),\theta} \in \mathbb{R}^m$ with $\vec{b}_i^{(n),\theta} := \tau_n(\phi_i^{(n)} \,;\, f^{(n),\theta}) + (\phi_i^{(n)} \,;\, g^{(n),\theta})_{\Gamma_N} + \{\underline{T}^{(n),\theta}\vec{u}^{(n-1)}\}_i$, $i = 1, \dots, m$. Note that the Neumann term $(\phi_i^{(n)} \,;\, g^{(n),\theta})_{\Gamma_N}$ is trivial for all nodes that do not lie on the Neumann boundary, that is for all i with $x_i \notin \Gamma_N$. Since $\underline{M}^{(n),\theta}$ is positive definite, it is regular. The term $\tau_n\theta(\underline{D}^{(n),\theta} + \underline{C}^{(n),\theta} + \underline{R}^{(n),\theta})$ may be regarded as a perturbation of $\underline{M}^{(n),\theta}$ with the perturbation variable $\tau_n\theta$. Though the matrices $\underline{D}^{(n),\theta}$, $\underline{C}^{(n),\theta}$ and $\underline{R}^{(n),\theta}$ may not be regular in general, $\underline{M}^{(n),\theta} + \tau_n\theta(\underline{D}^{(n),\theta} + \underline{C}^{(n),\theta} + \underline{R}^{(n),\theta})$ is regular for $\tau_n\theta$ sufficiently small. Thus, (2.32) has a unique solution. Note, that with the above restriction on the size of $\tau_n\theta$ we may never reach the end of the time interval T.
Let $u_h^{(n)} \in \mathcal{S}_D^1(\mathcal{T}^{(n)})$ for $n = 0, \dots, N$ be the discrete solution of (2.31) and (2.29). We then define $u_{h,\tau}$,

$$u_{h,\tau}\big|_{I_n} := \frac{t - t_{n-1}}{\tau_n}u_h^{(n)} + \frac{t_n - t}{\tau_n}u_h^{(n-1)}$$

for $n = 1, \dots, N$, i.e. $u_{h,\tau}$ is the affine interpolant with respect to time of the quasistatic solutions $u_h^{(n)}$.

2.3.4 Convergence

We now follow the arguments given in Verfürth, [17, Section III.4]. For simplicity we assume that there is no convection term, i.e. $\vec{c} = 0$, and that the data functions D and r are time-independent. For the next theorem we need the following notation:

$$\|D\|_{C^k} := \max_{|\alpha|\le k}\max_{x\in\mathbb{R}^2}\|\partial^\alpha D(x)\|$$

where $\|\cdot\|$ denotes any norm on $\mathbb{R}^{2\times 2} \simeq \mathbb{R}^4$. In order to obtain meaningful estimates, all involved right hand sides have to be bounded, therefore we will require additional regularity restrictions for u.

Theorem 2.6 (Verfürth, [17, Theorem III.4.11, p.131]). *Let u be the solution of (0.1) and $u_{h,\tau}$ be the solution of the discrete problem (2.31). Let us consider N uniform time steps with time step size τ and uniform spatial refinement with the common diameter of the elements h. Then, the following statements hold:*
If $\theta = 1/2$, then

$$\max_{0\le n\le N} \|u(\cdot,n\tau) - u_{h,\tau}(\cdot,n\tau)\| \lesssim (\tau^2+h)\max\{\max_{0\le k\le 3}\|\partial_t^k u\|_{\mathcal{C}(\overline{\Omega}\times[0,T],\mathbb{R})}, \max_{0\le|\alpha|\le 3}\|\partial^\alpha u\|_{\mathcal{C}(\overline{\Omega}\times[0,T],\mathbb{R})}\}\max\{1,\|D\|_{C^2}\}. \tag{2.33}$$

If $\theta \in (1/2,1]$, then

$$\max_{0\le n\le N} \|u(\cdot,n\tau) - u_{h,\tau}(\cdot,n\tau)\| \lesssim (\tau+h)\max\{\max_{0\le k\le 2}\|\partial_t^k u\|_{\mathcal{C}(\overline{\Omega}\times[0,T],\mathbb{R})}, \max_{0\le|\alpha|\le 3}\|\partial^\alpha u\|_{\mathcal{C}(\overline{\Omega}\times[0,T],\mathbb{R})}\}\max\{1,\|D\|_{C^2}\}. \tag{2.34}$$

■

Note that these estimates do not say anything about the quality of the approximation between two time steps. In the numerical experiments, we thus expect the convergence rate $\mathcal{O}(\tau_{max}^2 + h_{max})$ for $\theta = 1/2$, and the convergence rate $\mathcal{O}(\tau_{max} + h_{max})$ for $\theta \in (1/2, 1]$, where the τ_{max}, h_{max} denote the maximal step size in time and the maximal space element diameter, respectively.

2.3.5 The Function *parabolicSolver*

The function *parabolicSolver* provides a solution of the system (2.32) for given $\mathcal{T}^{(n-1)}$, $\mathcal{T}^{(n)}$, $u_{h,\tau}(t_{n-1})$, t_{n-1}, and t_n. From now on, we denote these quantities by their implemented data counterparts *OldGrid*, *Grid*, *Uold*, *Told*, and T. The main computational work is to assemble the matrices $M := \underline{M}^{(n),\theta}$, $H := \underline{D}^{(n),\theta} + \underline{C}^{(n),\theta} + \underline{R}^{(n),\theta}$ and the right hand side $\vec{b}^{(n),\theta}$, including the effect of the transfer matrix $\underline{T}^{(n),\theta}$.
If *Grid* and *OldGrid* coincide, we use stored data to simplify the assembling process. Hence, the variable AssemblyIdentifier is introduced to indicate, which of the data may be reused.

```
0165   %AssemblyIdentifier minimizes computation by indicating already stored
0166   %data
0167   AssemblyIdentifier = zeros(1,3);
0168   M = OldGrid.M;
0169   H = OldGrid.H;
0170   b = OldGrid.b;
0171   DoubleArea = OldGrid.DoubleArea;
0172   if GridsAreEqual
0173       if ~isempty(M)
0174           AssemblyIdentifier(1) = 1;
0175       end
0176       if (Options.Dependence.D<=0 || Options.Dependence.D==2) ...
0177           && (Options.Dependence.C<=0 || Options.Dependence.C==2) &&...
```

```
0178            (Options.Dependence.R<=0 || Options.Dependence.R==2)
0179            if ~isempty(H)
0180                AssemblyIdentifier(2) = 1;
0181            end
0182        end
0183        if Options.Dependence.F<=0 || Options.Dependence.F==2
0184            if ~isempty(b)
0185                AssemblyIdentifier(3) = 1;
0186            end
0187        end
0188    end
0189    if ~isempty(Grid.M)
0190        AssemblyIdentifier(1) = 1;
0191        M = Grid.M;
0192        DoubleArea = Grid.DoubleArea;
0193    end
0194    if ~isempty(Grid.H) ...
0195            && (Options.Dependence.D<=0 || Options.Dependence.D==2) ...
0196            && (Options.Dependence.C<=0 || Options.Dependence.C==2) &&...
0197            (Options.Dependence.R<=0 || Options.Dependence.R==2)
0198        AssemblyIdentifier(2) = 1;
0199        H = Grid.H;
0200    end
0201    if ~isempty(Grid.b)...
0202                        && (Options.Dependence.F<=0 || Options.Dependence.F==2)
0203        AssemblyIdentifier(3) = 1;
0204        b = Grid.b;
0205    end
```

Now, we identify the nodes which are currently used in *Grid* as well as the Dirichlet boundary nodes $\mathcal{N}_D^{(n)}$ and the free nodes $\mathcal{N}_f^{(n)}$.

```
0212    UsedNodes = setdiff(1:size(C4N,1),Grid.Unused.C4N);
0213    BoundaryNodes = unique(N4D);
0214    FreeNodes = setdiff(UsedNodes,BoundaryNodes);
0220    nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0221    U = zeros(size(C4N,1),1);
```

Next, we build the index vectors for the sparse representation of the system matrices, which is to be used with the MATLAB built-in function sparse, c.f. page 19. These indices are defined in $N4E$, which stores the nodes of *Grid*

```
0223    %Assembly
0224    %Prelocating vectors for assembly
0225    a = [N4E(nze,1) N4E(nze,1) N4E(nze,1) N4E(nze,2) N4E(nze,2)...
0226         N4E(nze,2) N4E(nze,3) N4E(nze,3) N4E(nze,3)]';
0227    I2 = reshape(a,9*nnz(nze),1);
0228    a = [N4E(nze,1:3) N4E(nze,1:3) N4E(nze,1:3)]';
0229    I1 = reshape(a,9*nnz(nze),1);
0230    LocMassMatrices = zeros(9*nnz(nze),1);
0231    Nnze = nnz(nze);
```

If no data can be reused, all matrices and the right hand side have to be assembled. For all $K \in \mathcal{T}^{(n)}$, we therefore compute the double areas $2|K|$ and the weighted gradients $2|K|\big(\nabla\phi_i|_K\big)$ of the basis functions for all vertices $i = 1, 2, 3$. The local contributions $(\phi_i\,;\,\phi_j)_K$ of the mass matrix M are obtained by multiplying the values for the reference element $\widehat{K} := \{(0,0);(1,0);(0,1)\}$, i.e.

$$\text{RefM} := (\widehat{\phi}_i\,;\,\widehat{\phi}_j)_{\widehat{K}} = \frac{1}{24}\begin{pmatrix}2 & 1 & 1\\ 1 & 2 & 1\\ 1 & 1 & 2\end{pmatrix} \text{ with } i,j = 1,2,3 \text{ and } \widehat{\phi}_1 = 1-x-y, \widehat{\phi}_2 = y, \widehat{\phi}_3 = y,$$

with the double areas. Here, for abbreviation, the functions ϕ_i, $i = 1, 2, 3$, are the nodal basis functions associated with the vertices of K. The values for the transport matrix H and the right hand side $\vec{b}$ are obtained by calling appropriate subfunctions, which are presented below.

```
0236   if ~nnz(AssemblyIdentifier)
0237
0238      %Vertices
0239      A = C4N(N4E(nze,1),:);
0240      B = C4N(N4E(nze,2),:);
0241      C = C4N(N4E(nze,3),:);
0242      %Assembly of areas
0243      DoubleArea = B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
0244                   -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0245      %Assembly of mass matrix
0246      LocMassMatrices = reshape(RefM'*DoubleArea',9*Nnze,1);
0247      %Assembly of gradients
0248      N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)];
0249      N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)];
0250      N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)];
0251      %Assembly of transport matrix
0252      LocHMatrices = ...
0253   stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options) ...
0254   + convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options) ...
0255   + reactionMatrix(ValR,A,B,C,DoubleArea,LocMassMatrices,Told,T,Driver,...
0256                                                                Options);
0257      %Assembly of RHS
0258      LocBs = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options);
0259
0260      M = sparse(I1,I2,LocMassMatrices,size(C4N,1),size(C4N,1));
0261      H = sparse(I1,I2,LocHMatrices,size(C4N,1),size(C4N,1));
0262      I1 = reshape(N4E(nze,1:3)',3*nnz(nze),1);
0263      b = sparse(I1,ones(nnz(I1),1),LocBs,size(C4N,1),1);
0264   end
```

The code for the transfer matrix $\underline{T}$ is similar[5]. If *OldGrid* and *Grid* coincide, there holds

$$\underline{T} = M - \tau_n(1-\theta)H.$$

Otherwise, the subfunction $transferMatrix$ computes the values of $\underline{T}$.

[5]To obtain the transfer matrix *parabolicSolver* calls *commonGrid*, c.f Section 3.8.1.

```
0348   %Tranfermatrix to old grid
0349   if GridsAreEqual
0350       Transma = M - (T-Told)*(1-Options.Theta)*H;
0351   else
0352
0353      %rows=new;columns=old
0354      [LocTransferMatrices,I1,I2] = transferMatrix(CGrid,C4N,N4E,...
0355            C4Nold,N4Eold,ValD,ValC,ValR,Told,T,Driver,Options);
0356      Transma =sparse(I1,I2,LocTransferMatrices,size(C4N,1),size(C4Nold,1));
0357   end
```

The Neumann data are treated differently, since the number of Neumann nodes is usually small and it does not influence the performance of the code

```
0337   %Neumann
0338   for j = 1 : size(N4N,1)
0339
0340       Locg = (Options.Theta*feval(Driver,'g(x,t)',...
0341                                                 sum(C4N(N4N(j,:),:))/2,T)...
0342              +(1-Options.Theta)*feval(Driver,'g(x,t)',...
0343                                               sum(C4N(N4N(j,:),:))/2,Told));
0344       b(N4N(j,:))= b(N4N(j,:)) + norm(C4N(N4N(j,1),:) ...
0345                                                - C4N(N4N(j,2),:))*Locg/2;
0346   end
```

Finally, we consider the Dirichlet boundary conditions, build the right hand side, and solve the linear system of equations

```
0361   %Boundary Condition
0362   U(BoundaryNodes) = feval(Driver,'u(x,t)',C4N(BoundaryNodes,:),T);
0364   %RHS
0365   b = Transma*Uold + (T-Told) * b - (M + (T-Told) * Options.Theta * H)*U;
0369   %prepare for solving
0370   H2 = M + (T-Told) * Options.Theta * H;
0371   H2 = H2(FreeNodes,FreeNodes);
0372   b2 = b(FreeNodes);
0373
0374   %Solving
0390       U(FreeNodes) = H2\b2;
0392
```

The subfunctions for space-time dependent data

For the computation of the values of the stiffness matrix $\underline{D}$, the convection matrix $\underline{C}$, the reaction matrix $\underline{R}$, and the right hand side $\vec{b}$, we use a quadrature rule which is specified in the function *options* and consequently in the structure Options (see Appendix A.8). Thus, if the data function D depends on time and space, we obtain the values for the stiffness matrix by its local entries

$$\underline{D}_{ij}|_K = (\nabla\phi_i \, ; \, D^{(n),\theta}\nabla\phi_j)_K \text{ with } i,j = 1,2,3.$$

```
0425 function StiffnessMatrix= ...
0426       stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options)
0428
0461           ValD = zeros(2*size(A,1),2);
0462
0463           for j=1:size(Options.QWeights,1)
0464
0465                QuadraturVertices = ...
0466                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0467                ValD = ValD + Options.Theta*Options.QWeights(j)*...
0468                                  feval(Driver,'D(x,t)',QuadraturVertices,T);
0469                if Options.Theta~=1
0470                    ValD = ValD + (1-Options.Theta)*Options.QWeights(j)*...
0471                              feval(Driver,'D(x,t)',QuadraturVertices,Told);
0472                end
0473           end
0474           ValD =reshape([ValD(1:size(A,1),1)';ValD(size(A,1)+1:end,1)';...
0475                          ValD(1:size(A,1),2)';ValD(size(A,1)+1:end,2)'],...
0476                                                          2,2*size(A,1))';
0477           ValD = blktridiag(ValD,size(A,1));
0478           N1Col = reshape(ValD*reshape(N1',2*size(A,1),1),2,size(A,1))';
0479           N2Col = reshape(ValD*reshape(N2',2*size(A,1),1),2,size(A,1))';
0480           N3Col = reshape(ValD*reshape(N3',2*size(A,1),1),2,size(A,1))';
0482
0483   a = [(N1(:,1).*N1Col(:,1) + N1(:,2).*N1Col(:,2))./DoubleArea,...
0484        (N2(:,1).*N1Col(:,1) + N2(:,2).*N1Col(:,2))./DoubleArea,...
0485        (N3(:,1).*N1Col(:,1) + N3(:,2).*N1Col(:,2))./DoubleArea,...
0486        (N1(:,1).*N2Col(:,1) + N1(:,2).*N2Col(:,2))./DoubleArea,...
0487        (N2(:,1).*N2Col(:,1) + N2(:,2).*N2Col(:,2))./DoubleArea,...
0488        (N3(:,1).*N2Col(:,1) + N3(:,2).*N2Col(:,2))./DoubleArea,...
0489        (N1(:,1).*N3Col(:,1) + N1(:,2).*N3Col(:,2))./DoubleArea,...
0490        (N2(:,1).*N3Col(:,1) + N2(:,2).*N3Col(:,2))./DoubleArea,...
0491        (N3(:,1).*N3Col(:,1) + N3(:,2).*N3Col(:,2))./DoubleArea];
0492   StiffnessMatrix = reshape(a',9*size(N1,1),1);
```

Note that $\nabla\phi_i$,$\nabla\phi_j$ are $\mathcal{T}^{(n)}$-piecewise constant. The local values of the convection matrix $\underline{C}$ are given by

$$\underline{C}_{ij}|_K = (\phi_i \, ; \, \vec{c}^{(n),\theta} \cdot \nabla\phi_j)_K \text{ with } i,j = 1,2,3.$$

Thus, the values of $\underline{C}$ are computed by

```
0496 function ConvectionMatrix = ...
0497                convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options)
0499
0500    ConvectionMatrix = zeros(9*size(N1,1),1);
0533           for j=1:size(Options.QWeights,1)
0534
0535                QuadraturVertices = ...
0536                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0537                ValC = Options.Theta*...
```

```
0538                                  feval(Driver,'c(x,t)',QuadraturVertices,T);
0539                 if Options.Theta~=1
0540                     ValC = ValC + (1-Options.Theta)*...
0541                                 feval(Driver,'c(x,t)',QuadraturVertices,Told);
0542                 end
0543                 a = [dot(N1,ValC,2)*Options.QBasisfct(j,1),...
0544                      dot(N1,ValC,2)*Options.QBasisfct(j,2),...
0545                      dot(N1,ValC,2)*Options.QBasisfct(j,3),...
0546                      dot(N2,ValC,2)*Options.QBasisfct(j,1),...
0547                      dot(N2,ValC,2)*Options.QBasisfct(j,2),...
0548                      dot(N2,ValC,2)*Options.QBasisfct(j,3),...
0549                      dot(N3,ValC,2)*Options.QBasisfct(j,1),...
0550                      dot(N3,ValC,2)*Options.QBasisfct(j,2),...
0551                      dot(N3,ValC,2)*Options.QBasisfct(j,3)];
0552                 ConvectionMatrix = ConvectionMatrix + ...
0553                                 Options.QWeights(j)*reshape(a',9*size(N1,1),1);
0554             end
```

According to

$$\underline{R_{ij}}|_K = (\phi_i \, ; \, r^{(n),\theta}\phi_j)_K \text{ with } i,j = 1,2,3,$$

the values of the reaction matrix $\underline{R}$ are analogously obtained.

```
0560 function ReactionMatrix = ...
0561      reactionMatrix(ValR,A,B,C,DoubleArea,MassMatrix,Told,T,Driver,Options)
0565
0587             ReactionMatrix = zeros(9*size(A,1),1);
0588             for j=1:size(Options.QWeights,1)
0589                 BasisPairs = Options.QBasisfct(j,:)'*Options.QBasisfct(j,:);
0590                 QuadraturVertices = ...
0591                        A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0592                 ValR = Options.Theta*...
0593                                  feval(Driver,'r(x,t)',QuadraturVertices,T);
0594                 if Options.Theta~=1
0595                     ValR = ValR + (1-Options.Theta)*...
0596                                 feval(Driver,'r(x,t)',QuadraturVertices,Told);
0597                 end
0598                 ValR = reshape(repmat(ValR.*DoubleArea,1,9)',9*size(A,1),1);
0599                 BasisPairs = repmat(reshape(BasisPairs,9,1),size(A,1),1);
0600                 ReactionMatrix = ReactionMatrix + ...
0601                                        Options.QWeights(j)*ValR.*BasisPairs;
0602             end
```

Finally, with

$$\vec{b_i}|_K = (\phi_i \, ; \, f^{(n),\theta})_K \text{ with } i = 1,2,3,$$

the values of the right hand side $\vec{b}$ yield

```
0607 function RHS = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options)
0609
```

```
0627            a = zeros(3,size(A,1));
0628            for j =1:size(Options.QWeights,1)
0629                a = a + ...
0630                Options.Theta*Options.QWeights(j)*Options.QBasisfct(j,:)'...
0631                 *(feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0632                                 (C-A)*Options.QNodes(j,2),T).*DoubleArea)';
0633                if Options.Theta~=1
0634                    a = a + (1-Options.Theta)*...
0635                             Options.QWeights(j)*Options.QBasisfct(j,:)'...
0636                     *(feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0637                              (C-A)*Options.QNodes(j,2),Told).*DoubleArea)';
0638                end
0639            end
0640            RHS = reshape(a,3*size(A,1),1);
```

The complete code of *parabolicSolver* can be found in the Appendix A.3.

Chapter 3

A Posteriori Error Estimation

Since we translated the problem (0.1) to a computable form in the preceding chapter, we now want to take a look on the error we made by discretizing time and space[1]. In this chapter we derive a residual-based a posteriori error estimate. After stating some essential equations used throughout the entire chapter, we introduce the residual – a decisive functional to test the quality of the approximation $u_{h,\tau}$. Its norm is equivalent to the error measured in the space-time Banach space $X(0,T)$, and we consequently decompose it in parts corresponding to the spatial and time, and data error, respectively. These entities are individually estimated and yield lower and upper bounds for the error. Then, we concentrate on specific cases, e.g. regimes with large convection, to finally derive an upper bound which is computable. At the end of this chapter, we also introduce an algorithm to obtain a common refinement $\mathcal{T}$ of two given regular triangulations $\mathcal{T}_1$ and $\mathcal{T}_2$, i.e. a regular triangulation $\widetilde{\mathcal{T}}$, such that for all $K_1 \in \mathcal{T}_1$ and $K_2 \in \mathcal{T}_2$ there holds $K_1 \cap K_2 \in \widetilde{\mathcal{T}}$.

3.1 Some Essential Equations

For preparation, we specify a major inequality valid in Sobolev spaces:

Lemma 3.1 (Poincaré inequality, Praetorius [13, p.61–65]). *For $\omega \subseteq \mathbb{R}^2$ a bounded Lipschitz domain and $v \in H^1$ with $\int_\omega v\,dx = 0$ the following inequality holds*

$$\|v\| \leq c_P \operatorname{diam}(\omega) \|\nabla v\| \tag{3.1}$$

with the Poincaré constant c_P that depends only on the shape of ω. ■

It should be noted, that in convex domains ω the Poincaré constant c_P equals $1/\pi$, c.f Bebendorf [4, Theorem 3.2, p.754]. Moreover, a similar estimate, where the condition $|\int_\omega v\,dx| = 0$ is replaced by $v|_{\Gamma_D} = 0$, is called Friedrichs inequality, c.f. Praetorius [13, p.64]. Again, the constant depends only on the shape of ω. Still, for abbreviation we denote the constants in both, the Poincaré and the Friedrichs inequality, by c_P.
The following inverse estimate is valid for any finite dimensional subspace of polynomials of $H^1(K)$. Unfortunately, there is no similar estimate, which holds for all functions of H^1.

[1] In the following we call the elements of a triangulation K instead of T to avoid a mix-up with the final time T.

Lemma 3.2 (Inverse Estimate, Ainsworth [1, Theorem 1.3,p.8]). *For $k \in \mathbb{N}$, an element $K \in \mathcal{T}^{(n)}$ and $p \in \mathcal{P}_k(K)$ a polynomial of degree k, we have the estimate*

$$\|\nabla p\|_K \lesssim h_K^{-1} \|p\|_K \tag{3.2}$$

with a constant that depends only on k and the ratio h_K/ρ_K. ■

In addition, we need an inequality to estimate the one dimensional edge-norm with the two dimensional area norm:

Lemma 3.3 (Trace Equality, Carstensen [7]). *For $K \subseteq \Omega$ a regular 2-simplex (triangle), $E \subset K$ an edge of K, and P the vertex of K which is not on E, the following equality holds for all $v \in H^1(K)$*

$$\frac{1}{|E|}\int_E v\,ds - \frac{1}{|K|}\int_K v\,dx = \frac{1}{2|K|}\int_K (x-P)\cdot\nabla v\,dx. \tag{3.3}$$

■

Corollary 3.4 (Trace Inequality). *Under the assumptions of the trace equality (Lemma 3.3), the following inequality holds*

$$\|v\|_E^2 \leq \frac{|E|}{|K|}(h_K\|v\nabla v\|_{L^1(K)} + \|v\|_K^2). \tag{3.4}$$

Proof. If $v \in H^1(K)$, then also $v^2 \in H^1(K)$, so we can insert v^2 in the trace equality (3.3) and obtain

$$\frac{1}{|E|}\int_E v^2\,ds = \frac{1}{|E|}\|v\|_E^2 = \frac{1}{2|K|}\int_K (x-P)\cdot(2v\nabla v)\,dx + \frac{1}{|K|}\int_K v^2\,dx.$$

Since $|x-P| \leq h_K$ we obtain

$$\|v\|_E^2 \leq \frac{|E|}{|K|}(h_K\|v\nabla v\|_{L^1(K)} + \|v\|_K^2).$$

■

The implemented error estimator depends on the ratios

$$\mu_K := \frac{h_K}{\rho_K}, \tag{3.5}$$

where ρ_K denotes the height of $K \in \mathcal{T}^{(n)}$. For each time step n with $1 \leq n \leq N$ we also need an additional regular triangulation $\widetilde{\mathcal{T}}^{(n)}$ where both of the discrete solutions u_h^{n-1} and u_h^n, which are generally computed for different triangulations, are defined. Thus, we consider the regular triangulations $\mathcal{T}^{(n)}$,$\mathcal{T}^{(n-1)}$ and the data functions to satisfy the additional conditions:

ASSUMPTION.

(i) **Shape-regularity**: For an element $K \in \mathcal{T}^{(n)}$ the ratio μ_K from (3.5) is uniformly bounded by the shape-regularity constant c_S with respect to all time steps, i.e

$$c_S := \sup_{0\leq n\leq N}\max_{K\in\mathcal{T}^{(n)}} \mu_K. \tag{3.6}$$

(ii) **Transition condition**: For all time steps, there is a admissible and regular triangulation $\widetilde{\mathcal{T}}^{(n)}$, which is a refinement of both $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$. We define the transition constant

$$c_T := \sup_{1\le n\le N} \max_{\tilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \max_{K\in\mathcal{T}^{(n)};\tilde{K}\subseteq K} \frac{h_K}{h_{\tilde{K}}}. \tag{3.7}$$

(iii) **Time-regularity**: For the data functions, there holds $\dot{D} \in L^2(0,T;L^2(\Omega)^{2\times 2})$, $\dot{\vec{c}} \in L^2(0,T;L^2(\Omega)^2)$, $\dot{r} \in L^2(0,T;L^2(\Omega))$, $\dot{f} \in L^2(0,T;L^2(\Omega))$ and $\dot{g} \in L^2(0,T;L^2(\Gamma_N))$.

Remark. Due to (1.1) and the considerations at the end of Section 1.2, shape regularity is guaranteed by the used refinement-coarsening strategy. Moreover, c_S is estimated in Lemma 1.1. Because of the nested red-refinement and red-coarsening strategy the transition condition is also assured, c.f. Lemma 3.25. The last assumption is required for the estimation of the error due to time discretization of the data, c.f. Section 3.6.2.

3.2 Equivalence of Error and Residual

For estimating the error, we denote a special linear functional, which depends on the discrete solution $u_{h,\tau}$.

Definition 15. For the discrete space-time solution $u_{h,\tau}$ we denote by $R(u_{h,\tau})$ the **residual** defined by

$$\begin{aligned}\langle R(u_{h,\tau})\ ;\ v\rangle :=& (f\ ;\ v) + (g\ ;\ v)_{\Gamma_N} - (\partial_t u_{h,\tau}\ ;\ v) - (D\nabla u_{h,\tau}\ ;\ \nabla v)\\ &- (\vec{c}\cdot\nabla u_{h,\tau}\ ;\ v) - (r u_{h,\tau}\ ;\ v) \text{ for all } v\in H^1_D \text{ and } t\in(0,T].\end{aligned} \tag{3.8}$$

The functional $R(u_{h,\tau}) \in L^2(0,T,H^{-1})$ is a function of t in $(0,T]$.

Recall, that $u_{h,\tau}$ is a piecewise affine function in time, which equals $u_h^{(n)}$ for $t = t_n$ with $1 \le n \le N$. To prove that the norm of the error and of the residual are equivalent we need to show that the space-time measure of the error $u - u_{h,\tau}$ with the graph norm defined in Definition 12

$$\|w\|_{X(0,t)} = \Big\{ \|w\|^2_{L^\infty(0,t;L^2(\Omega))} + \|w\|^2_{L^2(0,t;H^1_D(\Omega))} + \int_0^t |\!|\!| \partial_t w + \vec{c}(x,t)\cdot\nabla w |\!|\!|^2_* ds \Big\}^{1/2}$$

is equivalent to the dual norm of the residual. Thus, we have to prove that the residual provides lower and upper bounds for the error.

Lemma 3.5 (Residual is lower bound for error). *For $v \in L^2(0,T;H^1_D)$ the following lower bound on the error holds*

$$\int_0^T \langle R(u_{h,\tau})\ ;\ v\rangle dt \le \sqrt{1+\kappa^2(\max\{c_r,1\})^2}\, \|u-u_{h,\tau}\|_{X(0,T)} \|v\|_{L^2(0,T,H^1_D)} \tag{3.9}$$

with the constants κ of (P2) and c_r of (P4).

Proof. Let $v \in L^2(0,T,H^1_D)$. We recall Definition 15 of the residual

$$\langle R(u_{h,\tau})\ ;\ v\rangle = (f\ ;\ v) + (g\ ;\ v)_{\Gamma_N} - (\partial_t u_{h,\tau}\ ;\ v) - (D\nabla u_{h,\tau}\ ;\ \nabla v) - (\vec{c}\cdot\nabla u_{h,\tau}\ ;\ v) - (r u_{h,\tau}\ ;\ v).$$

The weak form (2.26) yields $(f\,;\,v)+(g\,;\,v)_{\Gamma_N}=(\partial_t u\,;\,v)+(D\nabla u\,;\,\nabla v)+(\vec{c}\cdot\nabla u\,;\,v)+(ru\,;\,v)$, so that

$$\langle R(u_{h,\tau})\,;\,v\rangle=(\partial_t(u-u_{h,\tau})\,;\,v)+(D\nabla(u-u_{h,\tau})\,;\,\nabla v)+(\vec{c}\cdot\nabla(u-u_{h,\tau})\,;\,v)+(r(u-u_{h,\tau})\,;\,v). \tag{3.10}$$

The estimate (2.17) implies

$$\begin{aligned}\langle R(u_{h,\tau})\,;\,v\rangle \leq&(\partial_t(u-u_{h,\tau})+\vec{c}\cdot\nabla(u-u_{h,\tau})\,;\,v)+\kappa\max\{c_r,1\}|\!|\!|u-u_{h,\tau}|\!|\!|\,|\!|\!|v|\!|\!|\\ \leq&|\!|\!|\partial_t(u-u_{h,\tau})+\vec{c}\cdot\nabla(u-u_{h,\tau})|\!|\!|_*|\!|\!|v|\!|\!|+\kappa\max\{c_r,1\}|\!|\!|u-u_{h,\tau}|\!|\!|\,|\!|\!|v|\!|\!|\\ \leq&\sqrt{1+\kappa^2(\max\{c_r,1\})^2}(|\!|\!|\partial_t(u-u_{h,\tau})+\vec{c}\cdot\nabla(u-u_{h,\tau})|\!|\!|_*+|\!|\!|u-u_{h,\tau}|\!|\!|)\,|\!|\!|v|\!|\!|.\end{aligned}$$

Integrating from 0 to T, Cauchy's inequality proves

$$\begin{aligned}\int_0^T\langle R(u_{h,\tau})\,;\,v\rangle ds\leq&\sqrt{1+\kappa^2(\max\{c_r,1\})^2}\Big\{\int_0^T\big(|\!|\!|\partial_t(u-u_{h,\tau})+\vec{c}\cdot\nabla(u-u_{h,\tau})|\!|\!|_*^2\\ &+|\!|\!|u-u_{h,\tau}|\!|\!|^2\big)dt\Big\}^{1/2}\Big\{\int_0^T|\!|\!|v|\!|\!|^2dt\Big\}^{1/2}\\ \leq&\sqrt{1+\kappa^2(\max\{c_r,1\})^2}\|u-u_{h,\tau}\|_{X(0,T)}\|v\|_{L^2(0,T,H^1_D)}.\end{aligned}$$

This concludes the proof. ■

Lemma 3.6 (Residual is Upper Bound for the Error). *We have the following upper bound for the error*

$$\begin{aligned}\|u-u_{h,\tau}\|_{X(0,t_n)}\leq&\Big\{2(1+\kappa^2(\max\{c_r,1\})^2)\|u_0-\Pi_0u_0\|^2\\ &+2(2+\kappa^2(\max\{c_r,1\})^2)\|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}\Big\}^{1/2}\end{aligned} \tag{3.11}$$

with the constants κ of (P2) and c_r of (P4). Moreover, u_0 is the initial function from (0.1) *and Π_0u_0 denotes its L^2-projection onto the space $\mathcal{S}^1_D(\mathcal{T}^{(0)})$.*

Proof. We choose an integer n between 1 and N with the corresponding time step t_n. With

$$\begin{aligned}\frac{1}{2}\frac{d}{dt}\|(u-u_{h,\tau})(\cdot,t)\|^2=&\frac{1}{2}\int_\Omega\frac{d}{dt}\Big((u-u_{h,\tau})(x,t)\Big)^2dx\\ =&\int_\Omega(u-u_{h,\tau})(x,t)\partial_t(u-u_{h,\tau})(x,t)\,dx\\ =&(\partial_t(u-u_{h,\tau})(\cdot,t)\,;\,(u-u_{h,\tau})(\cdot,t))\end{aligned}$$

and the ellipticity (2.12), we obtain

$$\frac{1}{2}\frac{d}{dt}\|(u-u_{h,\tau})(\cdot,t)\|^2+|\!|\!|(u-u_{h,\tau})(\cdot,t)|\!|\!|^2\leq\langle R(u_{h,\tau})\,;\,(u-u_{h,\tau})(\cdot,t)\rangle, \tag{3.12}$$

where the latter inequality follows just by Definition 15. With Young's inequality[2] we obtain

$$\begin{aligned}\frac{1}{2}\frac{d}{dt}\|(u-u_{h,\tau})(\cdot,t)\|^2+|\!|\!|(u-u_{h,\tau})(\cdot,t)|\!|\!|^2\leq&|\!|\!|R(u_{h,\tau})|\!|\!|_*|\!|\!|u-u_{h,\tau}|\!|\!|\\ \leq&\frac{1}{2}|\!|\!|R(u_{h,\tau})|\!|\!|_*^2+\frac{1}{2}|\!|\!|(u-u_{h,\tau})(\cdot,t)|\!|\!|^2.\end{aligned}$$

[2]Young's inequality: $2ab\leq a^2+b^2$ for $a,b\in\mathbb{R}$.

Integrating this estimate from 0 to $t \leq t_n$ implies

$$\underbrace{\|(u-u_{h,\tau})(\cdot,t)\|^2}_{=(*)} - \|u_0 - \Pi_0 u_0\|^2 + \underbrace{\int_0^t |\!|\!| (u-u_{h,\tau}(\cdot,s))|\!|\!|^2 ds}_{=(**)} \leq \|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})} \tag{3.13}$$

with the initial function $u_0 = u(\cdot,0)$ and its L^2-projection $\Pi_0 u_0 = u_{h,\tau}(x,0)$. Dropping nonnegative contributions on the left side of (3.13) only decreases it. Thus, we drop $(*)$ to obtain

$$\|(u-u_{h,\tau})\|^2_{L^2(0,t_n,H_D^1)} \leq \|u_0 - \Pi_0 u_0\|_0^2 + \|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}, \tag{3.14}$$

and we drop $(**)$ to obtain

$$\|(u-u_{h,\tau})\|^2_{L^\infty(0,t_n,L^2)} \leq \|u_0 - \Pi_0 u_0\|_0^2 + \|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})} \tag{3.15}$$

for $t = t_n$. Recalling the definition of the graph norm $\|\cdot\|_{X(0,t_n)}$, c.f. Definition 12, we still need to estimate $(\partial_t(u-u_{h,\tau}) + \vec{c}\cdot\nabla(u-u_{h,\tau})(.,t)\ ;\ v)$. Equation (3.10) reads

$$\begin{aligned}(\partial_t(u-u_{h,\tau})\ ;\ v) + (\vec{c}\cdot\nabla(u-u_{h,\tau})\ ;\ v) =& \langle R(u_{h,\tau})\ ;\ v\rangle - (D\nabla(u-u_{h,\tau})\ ;\ \nabla v)\\ &- (r(u-u_{h,\tau})\ ;\ v),\end{aligned}$$

With the estimate (2.17) this yields

$$(\vec{c}\cdot\nabla(u-u_{h,\tau})\ ;\ v) + (r(u-u_{h,\tau})\ ;\ v) \leq |\!|\!| R(u_{h,\tau}) |\!|\!|_* |\!|\!| v |\!|\!| + \kappa \max\{c_r,1\} |\!|\!| u-u_{h,\tau} |\!|\!| |\!|\!| v |\!|\!|,$$

Hence,

$$\begin{aligned}|\!|\!| \partial_t(u-u_{h,\tau}) + \vec{c}\cdot\nabla(u-u_{h,\tau}) |\!|\!|_*^2 \leq& \Big\{ |\!|\!| R(u_{h,\tau}) |\!|\!|_* + \kappa\max\{c_r,1\} |\!|\!| u-u_{h,\tau} |\!|\!| \Big\}^2\\ \leq& 2 |\!|\!| R(u_{h,\tau}) |\!|\!|_*^2 + 2\kappa^2 (\max\{c_r,1\})^2 |\!|\!| u-u_{h,\tau} |\!|\!|^2.\end{aligned}$$

Integration from 0 to t_n implies

$$\begin{aligned}&\underbrace{\int_0^{t_n} |\!|\!| \partial_t(u-u_{h,\tau}) + \vec{c}\cdot\nabla(u-u_{h,\tau}) |\!|\!|_*^2 dt}_{= \|\partial_t(u-u_{h,\tau}) + \vec{c}\cdot\nabla(u-u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}}\\ &\qquad \leq 2 \underbrace{\int_0^{t_n} |\!|\!| R(u_{h,\tau}) |\!|\!|_*^2 dt}_{= \|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}} + 2\kappa^2(\max\{c_r,1\})^2 \underbrace{\int_0^{t_n} |\!|\!| u-u_{h,\tau} |\!|\!|^2 dt}_{= \|u-u_{h,\tau}\|^2_{L^2(0,t_n,H_D^1)}}\end{aligned}$$

and thus, with estimate (3.14) for the last term

$$\begin{aligned}&\|\partial_t(u-u_{h,\tau}) + \vec{c}\cdot\nabla(u-u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}\\ &\quad \leq 2\kappa^2(\max\{c_r,1\})^2 \|u_0 - \Pi_0 u_0\|_0^2 + 2(1+\kappa^2(\max\{c_r,1\})^2) \|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}.\end{aligned} \tag{3.16}$$

We now combine (3.14)-(3.16) and obtain

$$\begin{aligned}
\|u-u_{h,\tau}\|^2_{X(0,t_n)} =& \underbrace{\operatorname*{ess\,sup}_{0\le t\le t_n}\|(u-u_{h,\tau})(\cdot,t)\|^2}_{(3.15)} + \underbrace{\|(u-u_{h,\tau})(\cdot,t)\|^2_{L^2(0,t_n,H^1_D)}}_{(3.14)}\\
&+\underbrace{\|\partial_t(u-u_{h,\tau})+\vec{c}\cdot\nabla(u-u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}}_{(3.16)}\\
\le&\|u_0-\Pi_0u_0\|^2+\|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}+\|u_0-\Pi_0u_0\|^2\\
&+\|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}+2\kappa^2(\max\{c_r,1\})^2\|u_0-\Pi_0u_0\|^2\\
&+2\big\{1+\kappa^2(\max\{c_r,1\})^2\big\}\|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}\\
\le&2\big\{1+\kappa^2(\max\{c_r,1\})^2\big\}\|u_0-\Pi_0u_0\|^2\\
&+2\big\{2+\kappa^2(\max\{c_r,1\})^2\big\}\|R(u_{h,\tau})\|^2_{L^2(0,t_n,H_D^{-1})}.
\end{aligned}$$

■

3.3 Decomposition of the Residual

In practice we want to control the spatial and temporal error separately. Moreover we want to control the errors for each element $K\in\mathcal{T}^{(n)}$ for a fixed time step $0<t_n\le T$. Now, we first decompose the residual and into spatial, temporal and data parts. Afterwards, we further decompose the spatial residual into element and edge residuals. For abbreviation we introduce the functions

$$\begin{aligned}
f_{h,\tau}:=&\widetilde{\Pi}_h^{(n)}f^{(n),\theta}=\theta f_h^{(n)}+(1-\theta)f_h^{(n-1)},\\
g_{h,\tau}:=&\widetilde{\Pi}_{h,\Gamma_N}^{(n)}g^{(n),\theta}=\theta g_h^{(n)}+(1-\theta)g_h^{(n-1)},
\end{aligned}\tag{3.17}$$

for $t\in(t_{n-1},t_n]$, where $\widetilde{\Pi}_h^{(n)}$, $\widetilde{\Pi}_{h,\Gamma_N}^{(n)}$ denote the L^2-projection onto the space $\mathcal{S}^1_D(\widetilde{\mathcal{T}}^{(n)})$ and the L^2-projection onto the space of Γ_N-traces of $\mathcal{S}^1_D(\widetilde{\mathcal{T}}^{(n)})$-functions, respectively. Note that $f_{h,\tau}$ and $g_{h,\tau}$ are piecewise constant with respect to time on each time interval $(t_{n-1},t_n]$. Throughout, $u_{h,\tau}$ denotes the discrete solution of (0.1).

Definition 16. We define the **spatial residual** $R_h(u_{h,\tau})$ via its duality pairing

$$\begin{aligned}
\langle R_h(u_{h,\tau})\,;\,v\rangle :=&(f_{h,\tau}\,;\,v)+(g_{h,\tau}\,;\,v)_{\Gamma_N}-\frac{1}{\tau_n}(u_h^{(n)}-u_h^{(n-1)}\,;\,v)\\
&-(D^{(n),\theta}\nabla(\theta u_h^{(n)}+(1-\theta)u_h^{(n-1)})\,;\,\nabla v)\\
&-(\vec{c}^{(n),\theta}\cdot\nabla(\theta u_h^{(n),\theta}+(1-\theta)u_h^{(n-1)})\,;\,v)\\
&-(r^{(n),\theta}(\theta u_h^{(n)}+(1-\theta)u_h^{(n-1)})\,;\,v)\quad\forall\,v\in H^1_D,\ \text{in}\ (t_{n-1},t_n]
\end{aligned}\tag{3.18}$$

with the time discretizations $D^{(n),\theta}:=\theta D^{(n)}+(1-\theta)D^{(n-1)}$, $\vec{c}^{(n),\theta}:=\theta\vec{c}^{(n)}+(1-\theta)\vec{c}^{(n-1)}$ and $r^{(n),\theta}:=\theta r^{(n)}+(1-\theta)r^{(n-1)}$, for each integer $1\le n\le N$. Note that the functional $R_h(u_{h,\tau})\in L^2(0,T,H^{-1})$ is constant with respect to time on each interval $(t_{n-1},t_n]$.

Remark. If we move all terms of the space-time discretization (2.31) to either side, we obtain the spatial residual. Therefore, we obtain the **Galerkin orthogonality**

$$\langle R_h(u_{h,\tau}) \,;\, v_h\rangle = 0 \quad \forall\, v_h \in S_D^1(\mathcal{T}^{(n)}). \tag{3.19}$$

Definition 17. We define the **temporal residual** $R_\tau(u_{h,\tau}) \in L^2(0,T,H^{-1})$ via its duality pairing

$$\begin{aligned}\langle R_\tau(u_{h,\tau}) \,;\, v\rangle :=& (D^{(n),\theta}\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)} - u_{h,\tau}) \,;\, \nabla v)\\ &+ (\vec{c}^{(n),\theta}\cdot\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)} - u_{h,\tau}) \,;\, v)\\ &+ (r^{(n),\theta}(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)} - u_{h,\tau}) \,;\, v) \quad \forall\, v \in H_D^1, \text{ in } (t_{n-1},t_n].\end{aligned}$$

Definition 18. We define the **data residual** $R_D(u_{h,\tau}) \in L^2(0,T,H^{-1})$ via its duality pairing

$$\begin{aligned}\langle R_D(u_{h,\tau}) \,;\, v\rangle :=& (f - f_{h,\tau}, v) + (g - g_{h,\tau}, v)_{\Gamma_N}\\ &+ ((D^{(n),\theta} - D)\nabla u_{h,\tau} \,;\, \nabla v)\\ &+ ((\vec{c}^{(n),\theta} - \vec{c})\cdot\nabla u_{h,\tau} \,;\, v)\\ &+ ((r^{(n),\theta} - r)u_{h,\tau} \,;\, v) \quad \forall\, v \in H_D^1, \text{ in } (t_{n-1},t_n].\end{aligned}$$

Note that the data residual corresponds to the error due to the θ-weighted time discretization of the data. The spatial discretization error is treated element- and edge-wise below, c.f. Definition 21.

Lemma 3.7 (Decomposition of the residual). *On each time interval $(t_{n-1},t_n]$, the residual can be decomposed as*

$$\langle R(u_{h,\tau}) \,;\, v\rangle = \langle R_\tau(u_{h,\tau}) \,;\, v\rangle + \langle R_h(u_{h,\tau}) \,;\, v\rangle + \langle R_D(u_{h,\tau}) \,;\, v\rangle \quad \forall v \in H_D^1. \tag{3.20}$$

Proof. For $v \in H_D^1$ we first sum up the spatial and temporal residual from the Definitions 16 and 17

$$\begin{aligned}\langle R_\tau(u_{h,\tau}) \,;\, v\rangle + \langle R_h(u_{h,\tau}) \,;\, v\rangle =& (f_{h,\tau}, v) + (g_{h,\tau}, v)_{\Gamma_N} - (\frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n}, v)\\ &- (D^{(n),\theta}\nabla u_{h,\tau} \,;\, \nabla v) - (\vec{c}^{(n),\theta}\cdot\nabla u_{h,\tau} \,;\, v) - (r^{(n),\theta}u_{h,\tau} \,;\, v).\end{aligned}$$

Combined with the data residual from the Definition 18, this yields

$$\begin{aligned}&\langle R_\tau(u_{h,\tau}) \,;\, v\rangle + \langle R_h(u_{h,\tau}) \,;\, v\rangle + \langle R_D(u_{h,\tau}) \,;\, v\rangle\\ &\quad = (f \,;\, v) + (g \,;\, v)_{\Gamma_N} - (\frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n}, v) - (D\nabla u_{h,\tau} \,;\, \nabla v) - (\vec{c}\cdot\nabla u_{h,\tau} \,;\, v) - (ru_{h,\tau} \,;\, v).\end{aligned}$$

Since the derivative $\partial_t u_{h,\tau}$ is piecewise constant and equals $(u_h^{(n)} - u_h^{(n-1)})/\tau_n$ on each time interval (t_{n-1},t_n), we obtain

$$\begin{aligned}&\langle R_\tau(u_{h,\tau}) \,;\, v\rangle + \langle R_h(u_{h,\tau}) \,;\, v\rangle + \langle R_D(u_{h,\tau}) \,;\, v\rangle\\ &\quad = (f \,;\, v) + (g \,;\, v)_{\Gamma_N} - (\partial_t u_{h,\tau} \,;\, v) - (D\nabla u_{h,\tau} \,;\, \nabla v) - (\vec{c}\cdot\nabla u_{h,\tau} \,;\, v) - (ru_{h,\tau} \,;\, v).\end{aligned}$$

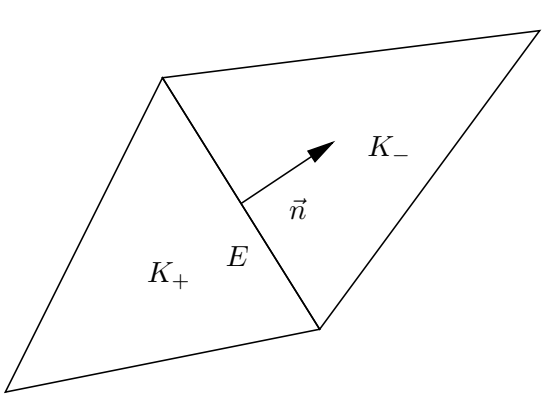


Figure 3.1: Notations for the definition of an edge jump from K_+ to K_-, c.f. Equation (3.21).

The expression on the right hand side of the previous equation equals $\langle R(u_{h,\tau}) \, ; \, v \rangle$, which concludes the proof. ■

In the implementation the error is controlled element-wise. Thus, we need a representation of the spatial residual $R_h(u_{h,\tau})$ that corresponds to a given triangulation $\widetilde{\mathcal{T}}^{(n)}$. For this purpose we introduce the element residual, which tests how "well" the problem (0.1) is solved on a specific element.

Definition 19. For $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$, we define the **element residual** by

$$R_{\widetilde{K}} := \Big[f_{h,\tau} - \frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n} + \operatorname{div}\{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} \\ - \vec{c}_h^{(n),\theta} \cdot \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) - r_h^{(n),\theta}(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) \Big] \Big|_{\widetilde{K}}$$

with $D_h^{(n),\theta}$, $\vec{c}_h^{(n),\theta}$ and $r_h^{(n),\theta}$ the L^2-projections on the finite element space $S_D^1(\widetilde{\mathcal{T}}^{(n)})$ of $D^{(n),\theta}$, $\vec{c}^{(n),\theta}$ and $r^{(n),\theta}$, respectively.

As we will see in the following, we also need an estimator how much the divergence differs from one element to another. This entity provides an approximation of the flux conservation for neighboring elements as well as for elements with an edge on the Neumann boundary.
Let K_+ and K_- be two neighboring elements that share the edge $E = K_+ \cap K_-$, c.f. Figure 3.1. Let q be in $H^1(K_+) \cap H^1(K_-)$, then there exist the restrictions $q|_{\partial K_+}$ and $q|_{\partial K_-}$ and we may define the **edge jump** of q from K_+ to K_- in the direction of the normal vector $\vec{n} := \vec{n}_{K_+}$ by

$$[q]_E := \big\{ q|_{\partial K_-} - q|_{\partial K_-} \big\}\big|_E. \tag{3.21}$$

Definition 20. For an edge $\widetilde{E} \in \mathcal{E}^{(n)}$, we define its **edge residual** by

$$R_{\widetilde{E}} := \begin{cases} -[\vec{n} \cdot \{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ g_{h,\tau} - \vec{n} \cdot \{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D. \end{cases}$$

The direction of the normal vector $\vec{n}$ defines the sign of the edge jump in (3.21), but not in the definition of the edge residual, since it includes $\vec{n}$ itself. Another choice of $\vec{n}$ gives twice the factor (-1) thereby leaving $R_{\widetilde{E}}$ unchanged.

Remark. The space-time discretizations $D_h^{(n),\theta}$, $\bar{c}_h^{(n),\theta}$, $r_h^{(n),\theta}$ and $f_{h,\tau}$ are locally linear on the elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$. Furthermore, $\nabla u_h^{(n)}$ and $\nabla u_h^{(n-1)}$ are $\widetilde{\mathcal{T}}^{(n)}$-piecewise constant. The mentioned discretizations are also locally linear, respectively constant, on the edges $\widetilde{E} \in \mathcal{E}^{(n)}$, as is $g_{h,\tau}$. Thus, for all time steps n, with $1 \leq n \leq N$, $R_{\widetilde{K}}$ and $R_{\widetilde{E}}$ are polynomials with at most degree 2 and 1, respectively.

In the definitions of the element residual and the edge residual, we used the L^2-projections on $S_D^1(\widetilde{\mathcal{T}}^{(n)})$ of the data functions. The thereby gained error is considered by the following definition.

Definition 21. Correspondingly to element and edge residual, we denote by

$$\begin{aligned} D_{\widetilde{K}} :=& \{ -\operatorname{div}\{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} \\ &+ (\bar{c}_h^{(n),\theta} - \bar{c}^{(n)}) \cdot \nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)}) + (r_h^{(n),\theta} - r^{(n),\theta})(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\}|_{\widetilde{K}} \end{aligned}$$

the **data residual of an element** $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ and by

$$D_{\widetilde{E}} := \begin{cases} [\vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ \vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D, \end{cases}$$

the **data residual of an edge** $\widetilde{E} \in \mathcal{E}^{(n)}$.

Lemma 3.8 (Decomposition of the spatial residual). *We consider $\widetilde{\mathcal{T}}^{(n)}$ to be a common refinement of $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$ and $\widetilde{\mathcal{E}}^{(n)}$ the set of edges of $\widetilde{\mathcal{T}}^{(n)}$. Then, we decompose the spatial residual by*

$$\langle R_h(u_{h,\tau}) \,;\, v\rangle = \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \{(R_{\widetilde{K}} \,;\, v)_{\widetilde{K}} + (D_{\widetilde{K}} \,;\, v)_{\widetilde{K}}\} + \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)}} \{(R_{\widetilde{E}} \,;\, v)_{\widetilde{E}} + (D_{\widetilde{E}} \,;\, v)_{\widetilde{E}}\} \quad \forall\, v \in H_D^1. \tag{3.22}$$

Proof. For $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$, the comparison of the element residuals, c.f. Definition 19, and the data residuals of an element, c.f. Definition 21, shows

$$\begin{aligned} R_{\widetilde{K}} + D_{\widetilde{K}} =& f_{h,\tau} - \frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n} + \operatorname{div}\{D^{(n),\theta}\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} \\ &- \bar{c}^{(n),\theta} \cdot \nabla(\theta u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)}) - r^{(n),\theta}(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)}) \end{aligned}$$

due to the linearity of all entities. Moreover, considering Definition 16 of the spatial residual we see, that the left hand and the right hand side of (3.22) equal in all but in the Neumann and divergence terms. Due to the linearity of the scalar product and Ω being the union of all $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$, the other terms can be directly identified as element–wise contributions to the right hand side of (3.22). For $v \in H_D^1$, we thus restrict to Neumann and divergence terms

$$\begin{aligned} &(g_{h,\tau} \,;\, v)_{\Gamma_N} - (D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)}) \,;\, \nabla v) \\ &= \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}_{\Gamma_N}^{(n)}} (g_{h,\tau} \,;\, v)_{\Gamma_N} - \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} (D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)}) \,;\, \nabla v)_{\widetilde{K}} \end{aligned} \tag{3.23}$$

with $\widetilde{\mathcal{E}}_{\Gamma_n}^{(n)}$ the set of edges of $\widetilde{\mathcal{T}}^{(n)}$ which are located on the Neumann boundary. With integration by parts, the second sum yields

$$\begin{aligned}
&- \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} (D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)}) \,;\, \nabla v)_{\widetilde{K}} \\
&= \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \Big\{ (\mathrm{div}\{D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})\} \,;\, v)_{\widetilde{K}} \\
&\qquad -(\vec{n}(x)\cdot(D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})) \,;\, v)_{\partial\widetilde{K}} \Big\}
\end{aligned} \tag{3.24}$$

with $\vec{n}(x)$ the outer normal vector on the edges of $\widetilde{K}$. The edges of an arbitrary element $\widetilde{K}_1 \in \widetilde{\mathcal{T}}^{(n)}$ may be located in the interior of Ω or on its boundary Γ. In case of the interior, there is exact one other element $\widetilde{K}_2 \in \widetilde{\mathcal{T}}^{(n)}$ which shares the same edge $\widetilde{E} \in \widetilde{\mathcal{E}}.^{(n)}$ Since the normal vector on the shared edge of these two element changes its sign, we obtain an edge jump $-[\vec{n}_{\widetilde{E}}\cdot(D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})]_{\widetilde{E}}$ – which equals $R_{\widetilde{E}} + D_{\widetilde{E}}$ on $\widetilde{E}$. On an edge of the Neumann boundary we have to consider the Neumann term of (3.23) which leads to

$$(g_{h,\tau} - \vec{n}_{\widetilde{E}}\cdot(D^{(n),\theta}(\theta\nabla u_h^{(n)} + (1-\theta)\nabla u_h^{(n-1)})) \,;\, v)_{\widetilde{E}},$$

whereas on the Dirichlet boundary v equals 0 and yields no contribution to the sum. In the definition of the edge residual, c.f. Definition 20, the value on the Dirichlet boundary is therefore arbitrary. Finally, replacing the sum of the element edges by a sum over edges in (3.23) and considering the data residual of an edge, c.f. Definition 21, proves the lemma. ■

3.4 Estimation of the Spatial Residual

As proven, the residual is equivalent to the error, c.f. Section 3.2. Analogously, we now want to show that the spatial residual itself is equivalent to an entity corresponding to the sum of an element, edge, and data residuals of an element and edge, respectively.

Definition 22. We denote by

$$\alpha_S := \min\{h_S\lambda^{-1/2}, \beta^{-1/2}\}$$

the **local parameter** of S. Here, S denotes an element $S = K \in \mathcal{T}^{(n)}$ or an edge $S = E \in \mathcal{E}^{(n)}$, h_S the diameter of the particular entity and λ, β are defined in the Assumptions $(P2)$ and $(P3)$.

Definition 23. For n between 1 and N and $\widetilde{\mathcal{T}}^{(n)}$ a corresponding common refinement of $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$ we denote by

$$\eta_h^{(n)} := \Big\{ \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \Big\}^{1/2}$$

the **Verfürth-type error estimator** with α the local parameter of Definition 22, R_K and R_E the element respectively edge residual of Definitions 19 and 20, and λ the constant of the Assumption $(P2)$. By

$$\Theta_h^{(n)} := \Big\{ \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|D_{\widetilde{K}}\|_{\widetilde{K}}^2 + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|D_{\widetilde{E}}\|_{\widetilde{E}}^2 \Big\}^{1/2}$$

we denote the **data error estimator**.

3.4.1 Reliability

Before we estimate the norm of the spatial residual $R_h(u_{h,\tau})$, we introduce a necessary preparation – an approximation operator which maps into the finite element space:

Definition 24. We denote by $I_h^{(n)}$ the **Clément operator** $I_h^{(n)} : H_D^1(\Omega) \to \mathcal{S}_D^1(\mathcal{T}^{(n)})$ defined by $I_h^{(n)} v := \sum_{i=1}^m \zeta_i \phi_i$, where $m := |\mathcal{N}^{(n)}|$. Here, ϕ_i denotes the nodal basis function on the i^{th} node x_i and the coefficients are defined by

$$\zeta_i := \begin{cases} \frac{1}{|\omega_{x_i}|} \int_{\omega_{x_i}} v \, dx & \text{if } x_i \in \mathcal{N}_f^{(n)}, \\ 0 & \text{if } x_i \in \mathcal{N}_D^{(n)} \end{cases}$$

with $\mathcal{N}_f^{(n)}$, $\mathcal{N}_D^{(n)}$ the set of all free nodes and set of all nodes on the Dirichlet boundary, respectively. The entity ω_{x_i} denotes the patch of the i^{th} node x_i. We will omit the index (n) if the time dependence is obvious.

Though by now, the Clément operator $I_h^{(n)}$ seems to be an arbitrary mapping into the finite element space $\mathcal{S}_D^1(\mathcal{T}^{(n)})$, the following lemma shows its decisive properties in the estimation of the spatial residual.

Lemma 3.9. *For all time steps n between 1 and N, all elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ with $\widetilde{K} \subseteq K \in \mathcal{T}^{(n)}$, all edges of $\widetilde{E}$ of $\widetilde{K}$ and all $v \in H_D^1$ the following estimates hold*

$$\|v - I_h^{(n)} v\|_{\widetilde{K}} \leq c_1 \alpha_{\widetilde{K}} |\!|\!| v |\!|\!|_{\omega_K}, \tag{3.25}$$

$$\|v - I_h^{(n)} v\|_{\widetilde{E}} \leq c_2 \lambda^{-\frac{1}{4}} \alpha_{\widetilde{E}}^{1/2} |\!|\!| v |\!|\!|_{\omega_K}. \tag{3.26}$$

The constants c_1, c_2 depend on the shape-regularity constant c_S of $\mathcal{T}^{(n)}$, the transition constant c_T, the Poincaré constant c_P, and on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$, but not on $\widetilde{K}$, $\widetilde{E}$, ω_K or v.

Proof. Let $v \in H_D^1$. To prove (3.25) we start with

$$\|v - I_h v\|_{\widetilde{K}} \leq \|v - I_h v\|_K.$$

since $\widetilde{K} \subseteq K$. We had to change to K, because the Clement operator is defined on $\mathcal{T}^{(n)}$ (see Definition 24). For K, only basis functions ϕ_i are non-trivial, which correspond to a vertex of K. Thus, the sum of the coefficients ζ_i is restricted to the set of nodes of K which we denote by $\mathcal{N}_K$. Furthermore, $\sum_{x_i \in \mathcal{N}_K} \phi_i = 1$ implies

$$\|v - I_h v\|_K = \left\| \sum_{x_i \in \mathcal{N}_K} (v - \zeta_i)\phi_i \right\|_K \leq \sum_{x_i \in \mathcal{N}_K} \|v - \zeta_i\|_K \underbrace{\|\phi_i\|_{L^\infty(K)}}_{=1}. \tag{3.27}$$

1. Claim. *There holds*

$$\|v - \zeta_i\|_K \leq 2\|v\|_K. \tag{3.28}$$

Due to Definition 24 we get

$$\begin{aligned} \|v - \zeta_i\|_K \leq \|v\|_K + \|\zeta_i\|_K = \|v\|_K + \left\| \frac{1}{|\omega_{x_i}|} \int_{\omega_{x_i}} v \, dx \right\|_K &\leq \|v\|_K + \frac{1}{|\omega_{x_i}|} \int_{\omega_{x_i}} \|v\|_K dx \\ &\leq 2\|v\|_K. \end{aligned}$$

□

2. Claim. *There holds*

$$\|v-\zeta_i\|_K \lesssim h_K\|\nabla v\|_{\omega_K} \tag{3.29}$$

with a constant that does neither depend on v nor on K, but on the shape-regularity constant c_S of $\mathcal{T}^{(n)}$ and the Poincaré constant c_P.

First, assume that ζ_i does not correspond to a node x_i on the Dirichlet boundary. Then, we need the Poincaré inequality (3.1) on ω_{x_i} as

$$\int_{\omega_i}(v-\zeta_i)\,dx = \int_{\omega_i} v\,dx - |\omega_i|\zeta_i = \int_{\omega_i} v\,dx - |\omega_i|\frac{1}{|\omega_i|}\int_{\omega_i} v\,dx = 0.$$

Therefore,

$$\|v-\zeta_i\|_K \leq \|v-\zeta_i\|_{\omega_i} \lesssim \underbrace{\operatorname{diam}(\omega_i)}_{\lesssim h_K}\|\nabla v\|_{\omega_i} \lesssim h_K\|\nabla v\|_{\omega_K}, \tag{3.30}$$

whereas the estimates include a constant that depends on the shape-regularity constant c_S of $\mathcal{T}^{(n)}$ and on the Poincaré constant c_P. If ζ_i is a coefficient corresponding to a Dirichlet node $\mathbf{x}_i \in \Gamma_D$, there holds $\zeta_i = 0$. Moreover, due to condition (iv) from Definition 1, there is an edge E of K, such that $v|_E = 0$ since $v \in H^1_D$. Thus, we may use Friedrichs inequality, c.f. Praetorius [13, p.64], which also implies $\|v\|_K \lesssim h_K\|\nabla v\|_{\omega_K}$. Therefore, estimate (3.29) follows. □

Now, (3.28) and (3.29) imply

$$\|v-\zeta_i\|_K \lesssim \min\{h_K\lambda^{1/2},\beta^{1/2}\}\Big(\lambda\|\nabla v\|^2_{\omega_{x_i}} + \beta\|v\|^2_{\omega_{x_i}}\Big)^{1/2} = \alpha_K|\!|\!|v|\!|\!|_{\omega_{x_i}}$$

since we may use either estimate for the left hand side. Here, α_K denotes the local parameter for K. Considering that there are three summands in (3.27), i.e. three vertices of K, gives

$$\|v-I_h^{(n)}v\|_{\widetilde{K}} \lesssim \alpha_K|\!|\!|v|\!|\!|_{\omega_K}.$$

Now, we use the transition condition (3.7) which bounds the ratio of the diameters of h_K and $h_{\widetilde{K}}$, such that $1 \leq c_T h_{\widetilde{K}}/h_K$ and therefore $\alpha_K \leq c_T\alpha_{\widetilde{K}}$ proving (3.25).
Equation (3.26) is proven by using the trace inequality (3.4) to estimate the edge norm

$$\begin{aligned}\|v-I_h^{(n)}v\|^2_{\widetilde{E}} &\leq \frac{|\widetilde{E}|}{|\widetilde{K}|}(h_{\widetilde{K}}\|(v-I_h^{(n)}v)\nabla(v-I_h^{(n)}v)\|_{L^1(\widetilde{K})} + \|v-I_h^{(n)}v\|^2_{\widetilde{K}})\\ &\leq \frac{2h_{\widetilde{E}}}{h_{\widetilde{K}}\rho_{\widetilde{K}}}(h_{\widetilde{K}}\|v-I_h^{(n)}v\|_K\|\nabla(v-I_h^{(n)}v)\|_K + \|v-I_h^{(n)}v\|^2_K).\end{aligned}$$

3. Claim. *There holds*

$$\|\nabla(v-I_h^{(n)}v)\|_K \lesssim \|\nabla v\|_K.$$

Since $\sum_{x_i \in \mathcal{N}_K} \phi_i = 1$ and $\nabla\phi_i \in \mathbb{R}^2$ we have that

$$\begin{aligned}\|\nabla(v - I_h^{(n)}v)\|_K &= \| \sum_{x_i\in\mathcal{N}_K} \nabla\big\{(v-\zeta_i)\phi_i\big\}\|_K \\ &\le \| \sum_{x_i\in\mathcal{N}_K} \phi_i\nabla v\|_K + \| \sum_{x_i\in\mathcal{N}_K} (v-\zeta_i)\nabla\phi_i)\|_K \\ &\le \|\nabla v\|_K + \sum_{x_i\in\mathcal{N}_K} \|v-\zeta_i\|_K|\nabla\phi_i|.\end{aligned}$$

Now, the Poincaré inequality bounds $\|v-\zeta_i\|_K \lesssim h_K\|\nabla v\|_K$ the same way as in the proof to the second claim. Note, that $\nabla\phi_i$ is constant on K and we obtain

$$\begin{aligned}h_K|\nabla\phi_i| \le c_S\rho_K|\nabla\phi_i| \le c_S|\nabla\phi_i|\Big\{\int_K dx\Big\}^{1/2} &= c_S\|\nabla\phi_i\|_K \\ &\le c_S h_K^{-1}\|\phi_i\|_K \\ &\le c_S h_K^{-1}\|\phi_i\|_{L^\infty(K)}\|1\|_K \\ &\le c_S\end{aligned}$$

where we also used the Inverse Estimate (3.2). Hence,

$$\|\nabla(v - I_h^{(n)}v)\|_K \lesssim \|\nabla v\|_K + 3\|\nabla v\|_K$$

since $|\mathcal{N}_K| = 3$, proving the claim. □

Consequently, we obtain

$$\|\nabla(v - I_h^{(n)}v)\|_K \lesssim \|\nabla v\|_K \le \lambda^{-1/2}\Big\{\lambda\|\nabla v\|_K^2 + \beta\|v\|_K^2\Big\}^{1/2}) \le \lambda^{-1/2}|\!|\!|v|\!|\!|_{\omega_K},$$

and with (3.25) we derive

$$\begin{aligned}\|v - I_h^{(n)}v\|_{\widetilde{E}}^2 &\le \frac{2h_{\widetilde{E}}}{h_{\widetilde{K}}\rho_{\widetilde{K}}}(h_{\widetilde{K}}\|v - I_h^{(n)}v\|_K\|\nabla(v - I_h^{(n)}v)\|_K + \|v - I_h^{(n)}v\|_K^2) \\ &\lesssim \frac{h_{\widetilde{E}}}{h_{\widetilde{K}}\rho_{\widetilde{K}}}(h_{\widetilde{K}}\lambda^{-1/2}\alpha_{\widetilde{K}} + \alpha_{\widetilde{K}}^2)|\!|\!|v|\!|\!|_{\omega_K}^2.\end{aligned}$$

4. Claim. *For the second factor of the previous equation we have the following estimate*

$$h_{\widetilde{K}}\lambda^{-1/2}\alpha_{\widetilde{K}} + \alpha_{\widetilde{K}}^2 \le 2h_{\widetilde{K}}\lambda^{-1/2}\alpha_{\widetilde{K}}.$$

Simple Analysis implies

$$h_{\widetilde{K}}\lambda^{-1/2}\alpha_{\widetilde{K}} + \alpha_{\widetilde{K}}^2 = h_{\widetilde{K}}\lambda^{-1/2}\alpha_{\widetilde{K}}(1 + \underbrace{h_{\widetilde{K}}^{-1}\lambda^{1/2}\min\{h_{\widetilde{K}}\lambda^{-1/2}, \beta^{-1/2}\}}_{\le 1}).$$

□

Since $h_{\widetilde{E}}/h_{\widetilde{K}} \le 1$ and $h_{\widetilde{K}}/\rho_{\widetilde{K}} = \mu_{\widetilde{K}}$ we get

$$\|v - I_h^{(n)}v\|_{\widetilde{E}}^2 \lesssim \lambda^{-1/2}\alpha_{\widetilde{K}}|\!|\!|v|\!|\!|_{\omega_K}^2$$

with a constant depending on shape regularity. The shape regularity also gives as an upper bound for the ratio $h_{\widetilde{K}}/h_{\widetilde{E}}$ which implies $\alpha_{\widetilde{K}} \lesssim \alpha_{\widetilde{E}}$ proving (3.26).

■

Lemma 3.10 (Reliability). *On each time interval $(t_{n-1}, t_n]$ with $1 \le n \le N$ the spatial residual is bounded from above by*

$$|||R_h(u_{h,\tau})|||_* \le c_{h,u}\{\eta_h^{(n)} + \Theta_h^{(n)}\} \tag{3.31}$$

with a constant $c_{h,u}$, which depends on the shape-regularity constant c_S from (3.6), on the transition constant c_T of (3.7), on the Poincaré constant c_P, and on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$.

Proof. For a n with $1 \le n \le N$ respectively a $v \in H^1_D$ and with Lemma 3.8

$$\langle R_h(u_{h,\tau}) \,;\, v\rangle = \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} (R_{\widetilde{K}} + D_{\widetilde{K}} \,;\, v)_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} (R_{\widetilde{E}} + D_{\widetilde{E}} \,;\, v)_{\widetilde{E}}$$

we get the following representation of $\langle R_h(u_{h,\tau}) \,;\, v\rangle$

$$\begin{aligned}\langle R_h(u_{h,\tau}) \,;\, v\rangle =& \langle R_h(u_{h,\tau}) \,;\, v - I_h^{(n)}v\rangle + \langle R_h(u_{h,\tau}) \,;\, I_h^{(n)}v\rangle \\ =& \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} (R_{\widetilde{K}} \,;\, v - I_h^{(n)}v)_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} (R_{\widetilde{E}} \,;\, v - I_h^{(n)}v)_{\widetilde{E}} \\ &+ \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} (D_{\widetilde{K}} \,;\, v - I_h^{(n)}v)_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} (D_{\widetilde{E}} \,;\, v - I_h^{(n)}v)_{\widetilde{E}}\end{aligned}$$

since $I_h^{(n)}v \in \mathcal{S}^1_D(\mathcal{T}) \subseteq H^1_D$ implying $v - I_h^{(n)}v \in H^1_D$ and $\langle R_h(u_{h,\tau}) \,;\, I_h^{(n)}v\rangle = 0$, because of the Galerkin's orthogonality (3.19). Furthermore, Cauchy's inequality implies

$$\begin{aligned}\langle R_h(u_{h,\tau}) \,;\, v\rangle \le& \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \|R_{\widetilde{K}}\|_{\widetilde{K}}\|v - I_h^{(n)}v\|_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \|R_{\widetilde{E}}\|_{\widetilde{E}}\|v - I_h^{(n)}v\|_{\widetilde{E}} \\ &+ \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \|D_{\widetilde{K}}\|_{\widetilde{K}}\|v - I_h^{(n)}v\|_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \|D_{\widetilde{E}}\|_{\widetilde{E}}\|v - I_h^{(n)}v\|_{\widetilde{E}}.\end{aligned}$$

The terms $\|v - I_h^{(n)}v\|_{\widetilde{K}}$ and $\|v - I_h^{(n)}v\|_{\widetilde{E}}$ can be estimated with (3.25) and (3.26), respectively

$$\begin{aligned}\langle R_h(u_{h,\tau}) \,;\, v\rangle \le& c_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}\|R_{\widetilde{K}}\|_{\widetilde{K}}|||v|||_{\omega_K} + c_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/4}\alpha_{\widetilde{E}}^{1/2}\|R_{\widetilde{E}}\|_{\widetilde{E}}|||v|||_{\omega_K} \\ &+ c_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}\|D_{\widetilde{K}}\|_{\widetilde{K}}|||v|||_{\omega_K} + c_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/4}\alpha_{\widetilde{E}}^{1/2}\|D_{\widetilde{E}}\|_{\widetilde{E}}|||v|||_{\omega_K}.\end{aligned}$$

where K denotes the element of $\mathcal{T}^{(n)}$ in which $\widetilde{K}$ is contained. Hence, with Cauchy's inequality

$$\begin{aligned}\langle R_h(u_{h,\tau})\,;\,v\rangle \leq &\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\omega_K}\right\}^{1/2}\cdot\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}}\|R_{\widetilde{K}}\|^2_{\widetilde{K}}\right\}^{1/2}\\ &+\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\omega_K}\right\}^{1/2}\cdot\left\{\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|^2_{\widetilde{E}}\right\}^{1/2}\\ &+\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\omega_K}\right\}^{1/2}\cdot\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}}\|D_{\widetilde{K}}\|^2_{\widetilde{K}}\right\}^{1/2}\\ &+\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\omega_K}\right\}^{1/2}\cdot\left\{\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|D_{\widetilde{E}}\|^2_{\widetilde{E}}\right\}^{1/2}.\end{aligned}$$

The number of elements $\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}$ contained in the patch ω_K is bounded, since the angles of all elements are bounded due to shape regularity and the transition condition. Thus, there is a constant that depends on c_S and c_T, such that

$$\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\omega_K} \lesssim \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} |\!|\!| v |\!|\!|^2_{\widetilde{K}} = |\!|\!| v |\!|\!|^2.$$

Consequently, with (2.16), we obtain

$$\begin{aligned}\langle R_h(u_{h,\tau})\,;\,v\rangle \lesssim &|\!|\!| v |\!|\!|\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}}\|R_{\widetilde{K}}\|^2_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|^2_{\widetilde{E}}\right\}^{1/2}\\ &+ |\!|\!| v |\!|\!|\left\{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}}\|D_{\widetilde{K}}\|^2_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|D_{\widetilde{E}}\|^2_{\widetilde{E}}\right\}^{1/2}\\ \lesssim &|\!|\!| v |\!|\!|\{\eta_h^{(n)} + \Theta_h^{(n)}\},\end{aligned}$$

The previous estimate holds for all $v\in H^1_D$ so we can divide by $|\!|\!| v |\!|\!|$ and take the supremum to get the dual norm on the left hand side. ■

3.4.2 Efficiency

For the efficiency of the spatial error estimator we assume for simplicity that the data D, $\vec{c}$ and r is constant with respect to space. Thus, there is no data error estimator $\Theta_h^{(n)}$ to deal with. While estimating entities for specific elements or edges, e.g. $|\!|\!|\cdot|\!|\!|_K$ for K an element of the triangulation $\mathcal{T}$, we expect the constant to be independent of K. A common technique of proving such inequalities uses a transformation of the integrals to a reference element[3]:

Definition 25. Set $K\in\mathcal{T}$, an element with the vertices $\{a,b,c\}$, $E_{a,b}\in\mathcal{E}$ and $\widehat{K}$ the **reference element**, defined by its vertices $\{(0,0),(1,0),(0,1)\}$, c.f. Figure 3.2(a). Then, we denote by Φ_K the affine mapping from $\widehat{K}$ to K, such that $(0,0)$, $(1,0)$ and $(0,1)$ are mapped to a, b and c, respectively, i.e.

$$\Phi_{K,E}(\hat{x},\hat{y}) := a + M_{K,E}\begin{pmatrix}\hat{x}\\ \hat{y}\end{pmatrix} \text{ with } M_{K,E} := \begin{pmatrix} b_1-a_1 & c_1-a_1\\ b_2-a_2 & c_2-a_2\end{pmatrix}$$

[3]This technique is commonly called scaling argument.

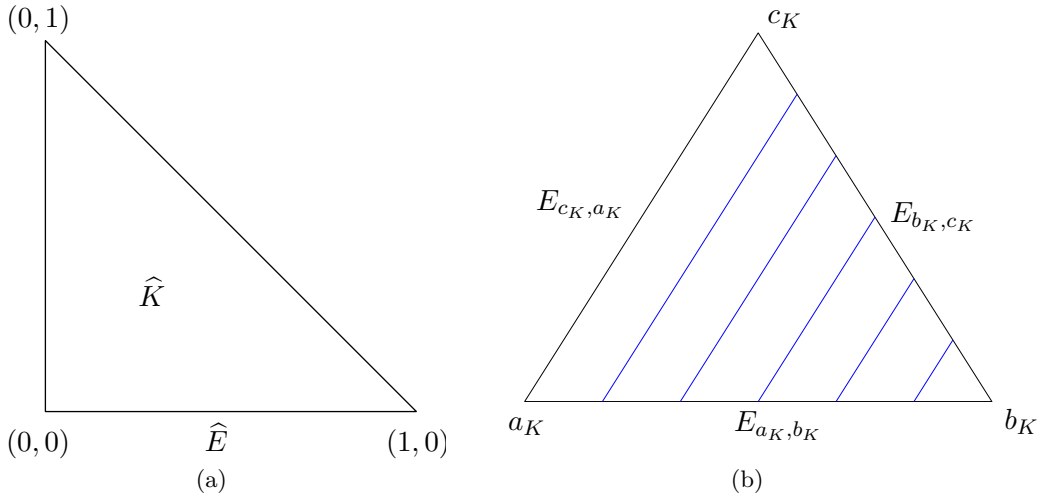


Figure 3.2: (a) The reference element $\widehat{K}$ and reference edge $\widehat{E}$. (b) Visualization of a quality of the extension operator $F^{\text{ext}}_{K,E}(\xi)$, c.f. Definition 26, for an element K and edge $E = E_{a_K,b_K}$. On all parallels of E_{c_K,a_K} (indicated as blues lines) the values $F^{\text{ext}}_{K,E}(\xi)(x,y)$ do not change.

for $(\hat{x},\hat{y}) \in \widehat{K}$. The Jacobian matrix of $\Phi_{K,E}$ is given by $D\Phi_{K,E} = M_{K,E}$.

We observe, that the determinant $|\det M_{K,E}| = 2|K|$ is bounded by the diameter of the element K

$$|\det M_{K,E}| \leq 2h_K^2. \tag{3.32}$$

Furthermore, $\Phi_{K,E}$ always maps the reference edge $\widehat{E} := \text{conv}\{(0,0),(1,0)\}$ to the introduced edge $E_{a,b}$ retaining the orientation. For the reference element $\widehat{K}$ the nodal basis functions can be easily identified, i.e. $\phi_{\{(0,0)\}}|_{\widehat{K}} := \widehat{\phi}_1 = (1-\hat{x}-\hat{y})$, $\phi_{\{(1,0)\}}|_{\widehat{K}} := \widehat{\phi}_2 = \hat{x}$ and $\phi_{\{(0,1)\}}|_{\widehat{K}} := \widehat{\phi}_3 = \hat{y}$, where $(\hat{x},\hat{y}) \in \widehat{K}$.
To measure functions defined on the edge E in the element K we need some kind of extension operator:

Definition 26. Let $\hat{\xi} \in L^\infty(\widehat{E})$ with $\widehat{E}$ the reference edge. Then, we denote by

$$\widehat{F}^{\text{ext}}(\hat{\xi})(x,y) := \hat{\xi}(x) \text{ for } (x,y) \in \widehat{K}$$

the **extension operator** for the reference element $\widehat{K}$. For $\xi \in L^\infty(E)$, E an edge of an element $K \in \mathcal{T}$, and $\Phi_{K,E}$ the mapping from Definition 25, the extension operator is then given by

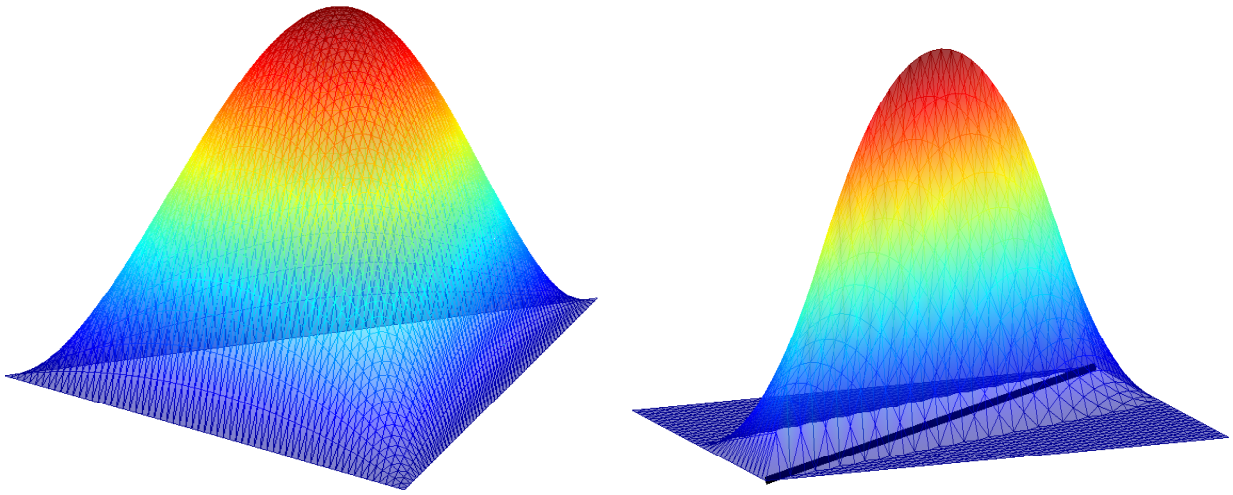
$$F^{\text{ext}}_{K,E}(\xi) := [\widehat{F}^{\text{ext}}(\xi \circ \Phi_{K,E})] \circ \Phi^{-1}_{K,E}.$$

We also define an extension operator for the patch ω_E of the edge E. If E is a boundary edge we define $F^{\text{ext}}_E := F^{\text{ext}}_{K,E}$. For an interior edge E, let K_1 and K_2 be the neighboring elements such that $E = K_1 \cap K_2$. Then, F^{ext}_E is defined by

$$F^{\text{ext}}_E\big|_{K_i} := F^{\text{ext}}_{K_i,E}$$

for $i = 1, 2$.

Because of the previous definition the values of $F^{\text{ext}}_{K,E}(\xi)(x,y)$ are the same on parallels of the



(a) (b)

Figure 3.3: (a) Bubble function ψ_K of an element K, c.f. Definition 27. (b) Bubble function ψ_E of an edge E, c.f. Definition 28. The edge E is indicated as thick black line and we choose $\Theta_E = 1/2$ in χ_E. In particular, ψ_E is trivial on half of the two neighboring elements that share E.

edge E_{c_K,a_K} for an element K, c.f. Figure 3.2(b).

Definition 27. For a $K \in \mathcal{T}$, we denote the **bubble function of an element** by

$$\psi_K := 27\phi_1\phi_2\phi_3 \tag{3.33}$$

where ϕ_i, for $i = 1, 2, 3$, are the basis function of the vertices of K, c.f. Figure 3.3(a).

The bubble function for an element is enforced to equal 1 in the barycenter of the element. Therefore, the coefficient 27 derives from this condition, since $\phi_i = 1/3$ for $i = 1, 2, 3$ in the barycenter of an element. Additionally, the support of ψ_K is contained in K, i.e. $\operatorname{supp}\psi_K \subset K$ for $K \in \mathcal{T}$, since $\operatorname{supp}\psi_K = \bigcap_{i=1}^{3} \operatorname{supp}\phi_i$. For scaling reasons which we describe in the following lemma, the definition of an analogous function for an edge is not that obvious. For an edge E and the reference element $\widehat{K}$ we need as auxiliary function the mapping

$$\chi_E(\hat{x}, \hat{y}) := \begin{pmatrix} 1 & 0 \\ 0 & \Theta_E \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{y} \end{pmatrix} \text{ for } (\hat{x}, \hat{y}) \in \widehat{K}, \tag{3.34}$$

where $\Theta_E := h_E^{-1}\lambda^{1/2}\alpha_E = \min\{1, h_E^{-1}\lambda^{1/2}\beta^{-1/2}\} \leq 1$. Note, that the image $\chi_E(\widehat{K}) \subseteq \widehat{K}$ and $\chi_E(\widehat{E}) = \widehat{E}$.

Definition 28. For $E \in \mathcal{E}$ the edge of one element K_1 or two elements K_1, K_2 we define the **bubble function of an edge** by

$$\psi_E|_{K_i} := \begin{cases} 4[\widehat{\phi}_1 \circ \chi_E^{-1} \circ \Phi_{K_i,E}^{-1}] \cdot \widehat{\phi}_2 \circ \Phi_{K_i,E}^{-1} & \text{for } (x,y) \in \chi_E(K_i) \\ 0 & \text{for } (x,y) \in K_i \setminus \chi_E(K_i) \end{cases} \tag{3.35}$$

with $i = 1$ or $i = 1, 2$ and $\widehat{\phi}_1, \widehat{\phi}_2$ denoting the first two basis functions for the reference element, c.f. Figure 3.3(b).

On an edge $E = E_{a,b}$ the bubble function equals the product of the basis functions of the vertices of E, i.e. $\psi_E|_E = 4\phi_a|_E\phi_b|_E$, since the term depending on Θ is trivial on E. Moreover, $\phi_i = 1/2$ for $i = 1, 2$ in the barycenter of an edge for all basis functions, implying the coefficient 4 with analogous considerations as for the bubble function for an element. Furthermore, the support of ψ_E is contained in the patch of the edge ω_E, i.e. $\operatorname{supp}\psi_E \subset \omega_E$ for $E \in \mathcal{E}$.
The decisive feature of the bubble functions is, that they are polynomials that "localize" integrals. That means, if a function defined on Ω is multiplied by ψ_K or ψ_E, the product is only non-trivial in the subset of the patch of an element or edge, respectively.

Lemma 3.11. *For every n with $1 < n \leq N$, all elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ and all polynomials $p \in \mathcal{P}_k(\widetilde{K})$, all edges $\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)}$ and all polynomials $p_{\widetilde{E}} \in \mathcal{P}_k(\widetilde{E})$ the following estimates hold*

$$(p\,;\psi_{\widetilde{K}}p)_{\widetilde{K}} \geq c_3\|p\|^2_{\widetilde{K}}, \tag{3.36}$$

$$|\!|\!|\psi_{\widetilde{K}}p|\!|\!|_{\widetilde{K}} \leq c_4\alpha_{\widetilde{K}}^{-1}\|p\|_{\widetilde{K}}, \tag{3.37}$$

$$(p_{\widetilde{E}}\,;\psi_{\widetilde{E}}p_{\widetilde{E}})_{\widetilde{E}} \geq c_5\|p_{\widetilde{E}}\|^2_{\widetilde{E}}, \tag{3.38}$$

$$\|\psi_{\widetilde{E}}F_E^{ext}(p_{\widetilde{E}})\|_{\omega_{\widetilde{E}}} \leq c_6\lambda^{1/4}\alpha_{\widetilde{E}}^{1/2}\|p_{\widetilde{E}}\|_{\widetilde{E}} \tag{3.39}$$

$$|\!|\!|\psi_{\widetilde{E}}F_E^{ext}(p_{\widetilde{E}})|\!|\!|_{\omega_{\widetilde{E}}} \leq c_7\lambda^{1/4}\alpha_{\widetilde{E}}^{-1/2}\|p_{\widetilde{E}}\|_{\widetilde{E}}, \tag{3.40}$$

with constants c_i, $i = 3, \ldots, 7$, that only depend on the polynomial degree k and on the ratio $\mu_{\widetilde{K}} = h_{\widetilde{K}}/\rho_{\widetilde{K}}$. The functions $\psi_{\widetilde{K}}, \psi_{\widetilde{E}}$ denote the bubble function for that specific element or edge, respectively.

Proof. The following proof uses norm equivalencies on the reference element $\widehat{K}$, which include constants that depend only on the polynomial degree of the polynomials $p \in \mathcal{P}_k(\widetilde{K})$ and $p_{\widetilde{E}} \in \mathcal{P}_k(\widetilde{E})$, respectively. Starting with (3.36) and integral by substitution with $\Phi_{K,E}$ from Definition 25 implies

$$\begin{aligned}(p\,;\psi_{\widetilde{K}}p)_{\widetilde{K}} = \int_{\widetilde{K}} p(x)^2\,\psi_{\widetilde{K}}(x)dx &= |\det M_{\widetilde{K},\widetilde{E}}| \int_{\widehat{K}} p(\Phi_{\widetilde{K},\widetilde{E}}(\hat{x}))^2\psi_{\widetilde{K}}(\Phi_{\widetilde{K},\widetilde{E}}(\hat{x}))d\hat{x}\\ &= |\det M_{\widetilde{K},\widetilde{E}}| \int_{\widehat{K}} \widehat{p}(\hat{x})^2\widehat{\psi}(\hat{x})d\hat{x}\\ &= |\det M_{\widetilde{K},\widetilde{E}}|(\widehat{p}\,;\widehat{\psi}\widehat{p})_{\widehat{K}}\end{aligned}$$

and analogous

$$\|p\|^2_{\widetilde{K}} = \int_{\widetilde{K}} p(x)^2dx = |\det M_{\widetilde{K},\widetilde{E}}| \int_{\widehat{K}} p(\Phi_{\widetilde{K},\widetilde{E}}(\hat{x}))^2d\hat{x} = |\det M_{\widetilde{K},\widetilde{E}}|\|\widehat{p}\|^2_{\widehat{K}},$$

where $\widehat{\psi} = \psi_{\widetilde{K}} \circ \Phi_{\widetilde{K},\widetilde{E}}$ is the bubble function for the reference element and $\widehat{p} := p \circ \Phi_{\widetilde{K},\widetilde{E}}$ defines a polynomial on $\widehat{K}$ with the same degree as p. By definition, the bubble function $\widehat{\psi}$ is only trivial at the edges of $\widehat{K}$, and positive elsewhere, c.f. Equation (3.33). Therefore, $(\widehat{p}\,;\widehat{\psi}\widehat{q})_{\widehat{K}} =: \langle\widehat{p}\,;\widehat{q}\rangle$ defines a scalar product and $(\widehat{p}\,;\widehat{\psi}\widehat{q})^{1/2}_{\widehat{K}}$ defines a norm on $\mathcal{P}_k(\widehat{K})$. Due to norm equivalency, there is a constant c_{ref}, such that

$$(p\,;\psi_{\widetilde{K}}p)_{\widetilde{K}} = |\det M_{K,E}|(\widehat{p}\,;\widehat{\psi}\widehat{p})_{\widehat{K}} \geq c^2_{\text{ref}}|\det M_{K,E}|\|\widehat{p}\|^2_{\widehat{K}} = c^2_{\text{ref}}\|p\|^2_{\widetilde{K}}.$$

Analogously, we prove inequality (3.38) by transformation of the edge $\widetilde{E}$ to the reference edge $\widehat{E}$.

Using the Inverse Estimate (3.2) for the left hand side of (3.37), we obtain

$$\begin{aligned}\|\psi_{\widetilde{K}}p\|_{\widetilde{K}} = \left\{\lambda\|\nabla(\psi_{\widetilde{K}}p)\|^2_{\widetilde{K}} + \beta\|\psi_{\widetilde{K}}p\|^2_{\widetilde{K}}\right\}^{1/2} \lesssim& \left\{\lambda h^{-2}_{\widetilde{K}}\|\psi_{\widetilde{K}}p\|^2_{\widetilde{K}} + \beta\|\psi_{\widetilde{K}}p\|^2_{\widetilde{K}}\right\}^{1/2}\\ \leq& \sqrt{2}\max\{h^{-1}_{\widetilde{K}}\lambda^{1/2}, \beta^{1/2}\}\|\psi_{\widetilde{K}}p\|_{\widetilde{K}}.\end{aligned}$$

From $\max\{a,b\} = 1/\min\{a^{-1}, b^{-1}\}$, we derive

$$\|\psi_{\widetilde{K}}p\|_{\widetilde{K}} \lesssim \alpha^{-1}_{\widetilde{K}}\|\psi_{\widetilde{K}}p\|_{\widetilde{K}} \leq \alpha^{-1}_{\widetilde{K}}\|p\|_{\widetilde{K}}$$

proving (3.37).
To prove the last two inequalities (3.39) and (3.40) we show them for a single element K contained in the patch $\omega_{\widetilde{E}}$. Since there belong at most two elements to $\omega_{\widetilde{E}}$ and the constants in the following estimates are independent of them, the general expressions follow.

1. Claim. *For the reference element $\widehat{K}$ with the edge $\widehat{E}$ from Definition 25 and a polynomial $\widehat{p} \in \mathcal{P}_k(\widehat{E})$ we have the estimate*

$$\|\widehat{F}^{ext}(\widehat{p})\|_{\widehat{K}} \lesssim \|\widehat{p}\|_{\widehat{E}} \tag{3.41}$$

with a constant that only depends on the polynomial degree k.

We have to show that $\|\widehat{F}^{\text{ext}}(\widehat{p})\|_{\widehat{K}}$ defines a norm on $\mathcal{P}_k(\widehat{E})$. By construction, $\widehat{F}^{\text{ext}}(\widehat{p}) = 0$ implies $\widehat{p} = 0$, which proves the definiteness. Additionally, the operator is linear, i.e. $\widehat{F}^{\text{ext}}(a\widehat{p}_1 + \widehat{p}_2)(x,y) = a\widehat{p}_1(x) + \widehat{p}_2(x) = a\widehat{F}^{\text{ext}}(\widehat{p}_1)(x,y) + \widehat{F}^{\text{ext}}(\widehat{p}_2)(x,y)$ for $\widehat{p}_1, \widehat{p}_2 \in \mathcal{P}(\widehat{E})$ and $a \in \mathbb{R}$, which yields homogeneity and triangle inequality. □

2. Claim. *For an element K with edge E and a polynomial $p_E \in \mathcal{P}_k(E)$ there holds*

$$\|F^{ext}_{K,E}(p_E)\|_K \lesssim h^{1/2}_E\|p_E\|_E \tag{3.42}$$

with a constant that only depends on the polynomial degree k and on the ratio $\mu_{\widetilde{K}} = h_{\widetilde{K}}/\rho_{\widetilde{K}}$.

Transformation to the reference element implies

$$\|F^{\text{ext}}_{K,E}(p_E)\|_K = |\det M_{K,E}|^{1/2}\|F^{\text{ext}}_{K,E}(p_E)\circ\Phi_{K,E}\|_{\widehat{K}}$$

with $\Phi_{K,E}$ and $M_{K,E}$ from Definition 25. Properties of the extension operator and the definition $\widehat{p}_E := p_E \circ \Phi_{K,E}$, give

$$\|[\widehat{F}^{\text{ext}}(p_E\circ\Phi_{K,E})]\circ\Phi^{-1}_{K,E}\circ\Phi_{K,E}\|_{\widehat{K}} = \|\widehat{F}^{\text{ext}}(\widehat{p})\|_{\widehat{K}}.$$

With (3.32) and (3.41), we conclude

$$\|F^{\text{ext}}_{K,E}(p_E)\|_K = |\det M_{K,E}|^{1/2}\|\widehat{F}^{\text{ext}}(\widehat{p})\|_{\widehat{K}} \lesssim h_K\|\widehat{p}\|_{\widehat{E}} = h_K h_E^{-1/2}\|p_E\|_E.$$

Now, $h_K = \mu_K\rho_K \leq \mu_K h_E$ proves the claim. □

3. Claim. *For an element K with edge E and polynomial $p_E \in \mathcal{P}_k(E)$, we have the following estimate*

$$\|\psi_E F^{ext}_{K,E}(p_E)\|_K \lesssim \Theta^{1/2}_E h^{1/2}_E\|p_E\|_E \tag{3.43}$$

with a constant that only depends on the polynomial degree k and on the ratio $\mu_{\widetilde{K}} = h_{\widetilde{K}}/\rho_{\widetilde{K}}$. Here, $\Theta_E := h_E^{-1}\lambda^{1/2}\alpha_E$.

Inserting the definition of the bubble function ψ_E, c.f. Definition 28, and transformation to the reference element $\widehat{K}$ gives

$$\begin{aligned}\|\psi_E F_{K,E}^{\mathrm{ext}}(p_E)\|_K^2 =& |\det(M_{K,E})| \int_{\widehat{K}} (\psi_E \circ \Phi_{K,E}(\hat{x},\hat{y}))^2 \cdot [F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E}(\hat{x},\hat{y}))]^2 d\hat{x}d\hat{y} \\ =& |\det(M_{K,E})| \int_{\chi_E(\widehat{K})} \left[4[\widehat{\phi}_1 \circ \chi_E^{-1}(\hat{x},\hat{y})] \cdot \widehat{\phi}_2(\hat{x},\hat{y})\right]^2 \\ & \cdot [F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E}(\hat{x},\hat{y}))]^2 d\hat{x}d\hat{y},\end{aligned}$$

since $\widehat{\psi} = \psi_E \circ \Phi_{K,E}$ is trivial on $\widehat{K}\setminus\chi(\widehat{K})$. The last two terms in the previous expression remain unchanged after a further transformation to the reference element[4] , since χ_E only affects the second coordinate and $\widehat{F}^{\mathrm{ext}}$ on the other hand depends only on $\hat{x}$, i.e. $\chi(\widehat{\phi}_2(\hat{x},\hat{y})) = \chi(\hat{x}) = \hat{x}$ and

$$\begin{aligned}F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E} \circ \chi_E(\hat{x},\hat{y})) = [\widehat{F}^{\mathrm{ext}}(p_E \circ \Phi_{K,E})](\chi_E(\hat{x},\hat{y})) &= [\widehat{F}^{\mathrm{ext}}(p_E \circ \Phi_{K,E})](\hat{x},\Theta_E\hat{y}) \\ &= [\widehat{F}^{\mathrm{ext}}(p_E \circ \Phi_{K,E})](\hat{x},\hat{y}) \\ &= F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E}(\hat{x},\hat{y})).\end{aligned}$$

Hence, with the determinant of the Jacobian matrix $\det D\chi_E = \Theta_E$,

$$\begin{aligned}&\|\psi_E F_{K,E}^{\mathrm{ext}}(p_E)\|_K^2 \\ &\quad = \Theta_E |\det(M_{K,E})| \int_{\widehat{K}} \Big[\underbrace{4\widehat{\phi}_1(\hat{x},\hat{y}) \cdot \widehat{\phi}_2(\hat{x},\hat{y})}_{\leq 1}\Big]^2 \cdot [F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E}(\hat{x},\hat{y}))]^2 d\hat{x}d\hat{y} \\ &\quad \leq \Theta_E |\det(M_{K,E})| \int_{\widehat{K}} [F_{K,E}^{\mathrm{ext}}(p_E)(\Phi_{K,E}(\hat{x},\hat{y}))]^2 d\hat{x}d\hat{y} \\ &\quad = \Theta_E \underbrace{\|F_{K,E}^{\mathrm{ext}}(p_E)\|_K^2}_{\overset{(3.42)}{\lesssim} h_E\|p_E\|_E^2}.\end{aligned}$$

□

In conclusion we have $\Theta_E^{1/2} h_E^{1/2} = h_E^{-1/2}\lambda^{1/4}\alpha_E^{1/2} h_E^{1/2} = \lambda^{1/4}\alpha_E^{1/2}$ proving (3.39).
For (3.40) we recall the definition of the local parameter of K, $\alpha_K = \min\{h_K\lambda^{-1/2}, \beta^{-1/2}\}$, the Inverse Estimate (3.2) and (3.39)

$$\begin{aligned}|||\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})|||_K =& \Big\{\lambda\|\nabla(\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}}))\|_K^2 + \beta\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})\|_K^2\Big\}^{1/2} \\ \lesssim& \Big\{\lambda h_K^{-2}\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})\|_K^2 + \beta\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})\|_K^2\Big\}^{1/2} \\ =& \{\lambda h_K^{-2} + \beta\}^{1/2}\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})\|_K \\ \leq& \sqrt{2}\max\{h_K^{-1}\lambda^{1/2}, \beta^{1/2}\}\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(p_{\widetilde{E}})\|_K \\ \lesssim& \alpha_K^{-1}\lambda^{1/4}\alpha_E^{1/2}\|p_{\widetilde{E}}\|_E \\ \lesssim& \lambda^{1/4}\alpha_E^{-1/2}\|p_{\widetilde{E}}\|_E.\end{aligned}$$

[4]One essential aspect of this consideration is, that $(\hat{x},\Theta_E\hat{y})$ still lies in $\widehat{K}$. Hence, the condition $\Theta_E \leq 1$ is indispensable.

For the final inequality we used

$$\alpha_K^{-1} = \max\{h_K^{-1}\lambda^{1/2}, \beta^{1/2}\} \leq \max\{h_E^{-1}\lambda^{1/2}, \beta^{1/2}\} = \alpha_E^{-1},$$

which concludes the proof. ■

Lemma 3.12 (Efficiency). *There are functions $w^{(n)} \in H^1_D$, such that on each time interval $(t_{n-1}, t_n]$ the spatial residual is bounded from below by*

$$\begin{aligned} \left(\eta_h^{(n)}\right)^2 \leq& \langle R_h(u_{h,\tau}) \,;\, w^{(n)} \rangle, \\ |\!|\!| w^{(n)} |\!|\!| \leq& c_{h,\ell}\eta_h^{(n)} \end{aligned} \tag{3.44}$$

with a constant $c_{h,\ell}$ that depends only on the shape regularity constant c_S and the transition constant c_T.

Proof. We define the function $w^{(n)}$ by

$$w^{(n)} := \gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \psi_{\widetilde{K}} R_{\widetilde{K}} + \gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \tag{3.45}$$

with constants γ_1 and γ_2 yet to be determined. For the square of the energy norm of this function we obtain

$$\begin{aligned} |\!|\!| w^{(n)} |\!|\!|^2 =& \lambda(w^{(n)} \,;\, w^{(n)}) + \beta(\nabla w^{(n)} \,;\, \nabla w^{(n)}) \\ =& \gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\widetilde{K}'\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \alpha_{\widetilde{K}'}^2 \Big\{ \lambda(\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \psi_{\widetilde{K}'} R_{\widetilde{K}'}) + \beta(\nabla\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \nabla\psi_{\widetilde{K}'} R_{\widetilde{K}'}) \Big\} \\ &+ 2\gamma_1\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \alpha_{\widetilde{K}}^2 \lambda^{-1/2} \alpha_{\widetilde{E}} \Big\{ \lambda(\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})) \\ &\qquad + \beta(\nabla\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})) \Big\} \\ &+ \gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1} \alpha_{\widetilde{E}} \alpha_{\widetilde{E}'} \Big\{ \lambda(\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \,;\, \psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})) \Big\} \end{aligned}$$

For $\widetilde{K} \neq \widetilde{K}'$, the support of $\psi_{\widetilde{K}}$ and $\psi_{\widetilde{K}'}$ are disjoint. Therefore, $(\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \psi_{\widetilde{K}'} R_{\widetilde{K}'}) = (\nabla\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \nabla\psi_{\widetilde{K}'} R_{\widetilde{K}'}) = 0$, and these contributions of the first sum vanish. Analogously, $(\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})) = (\nabla\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})) = 0$, if E is no edge of K and $(\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \,;\, \psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})) = (\nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \,;\, \nabla\psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})) = 0$ if $\omega_E \cap \omega_{E'} = \emptyset$. Altogether, we obtain

$$\begin{aligned} |\!|\!| w^{(n)} |\!|\!|^2 =& \gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^4 |\!|\!| \psi_{\widetilde{K}} R_{\widetilde{K}} |\!|\!|_{\widetilde{K}}^2 \\ &+ 2\gamma_1\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\widetilde{K}\neq\emptyset}} \alpha_{\widetilde{K}}^2 \lambda^{-1/2} \alpha_{\widetilde{E}} \Big\{ \lambda(\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}))_{\widetilde{K}} \\ &\qquad + \beta(\nabla\psi_{\widetilde{K}} R_{\widetilde{K}} \,;\, \nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}))_{\widetilde{K}} \Big\} \\ &+ \gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\substack{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}\neq\emptyset}} \lambda^{-1} \alpha_{\widetilde{E}} \alpha_{\widetilde{E}'} \Big\{ \lambda(\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \,;\, \psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'}))_{\omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}} \\ &\qquad + \beta(\nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}) \,;\, \nabla\psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'}))_{\omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}} \Big\}. \end{aligned}$$

Due to Cauchy's inequality this yields

$$\begin{aligned}
\|\!|w^{(n)}|\!\|^2 \leq & \gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^4 \|\!|\psi_{\widetilde{K}} R_{\widetilde{K}}|\!\|_{\widetilde{K}}^2 \\
& + 2\gamma_1\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\widetilde{K}\neq\emptyset}} \alpha_{\widetilde{K}}^2 \lambda^{-1/2} \alpha_{\widetilde{E}} \Big\{ \lambda \|\psi_{\widetilde{K}} R_{\widetilde{K}}\|_{\widetilde{K}} \|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\widetilde{K}} \\
& \qquad + \beta \|\nabla\psi_{\widetilde{K}} R_{\widetilde{K}}\|_{\widetilde{K}} \|\nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\widetilde{K}} \Big\} \\
& + \gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\substack{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}\neq\emptyset}} \lambda^{-1} \alpha_{\widetilde{E}} \alpha_{\widetilde{E}'} \Big\{ \lambda \|\psi_{\widetilde{E}} R_{\widetilde{E}}\|_{\omega_{\widetilde{E}}} \|\psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})\|_{\omega_{\widetilde{E}'}} \\
& \qquad + \beta \|\nabla\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\omega_{\widetilde{E}}} \|\nabla\psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})\|_{\omega_{\widetilde{E}'}} \Big\}.
\end{aligned}$$

With

$$\lambda ab + \beta cd = \begin{pmatrix} \lambda^{1/2} a \\ \beta^{1/2} c \end{pmatrix} \cdot \begin{pmatrix} \lambda^{1/2} b \\ \beta^{1/2} d \end{pmatrix} \overset{\text{Cauchy}}{\leq} \left\| \begin{pmatrix} \lambda^{1/2} a \\ \beta^{1/2} c \end{pmatrix} \right\| \left\| \begin{pmatrix} \lambda^{1/2} b \\ \beta^{1/2} d \end{pmatrix} \right\|$$

and the definition of the energy norm, c.f. Definition 9, we arrive at

$$\begin{aligned}
\|\!|w^{(n)}|\!\|^2 \leq & \gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^4 \|\!|\psi_{\widetilde{K}} R_{\widetilde{K}}|\!\|_{\widetilde{K}}^2 \\
& + 2\gamma_1\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\widetilde{K}\neq\emptyset}} \alpha_{\widetilde{K}}^2 \lambda^{-1/2} \alpha_{\widetilde{E}} \|\!|\psi_{\widetilde{K}} R_{\widetilde{K}}|\!\|_{\widetilde{K}} \|\!|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})|\!\|_{\widetilde{K}} \\
& + \gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\substack{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}\neq\emptyset}} \lambda^{-1} \alpha_{\widetilde{E}} \alpha_{\widetilde{E}'} \|\!|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})|\!\|_{\omega_{\widetilde{E}}} \|\!|\psi_{\widetilde{E}'} F_{E'}^{\mathrm{ext}}(R_{\widetilde{E}'})|\!\|_{\omega_{\widetilde{E}'}}.
\end{aligned}$$

Now, we use Lemma 3.11 and estimate $\|\!|\psi_{\widetilde{K}} R_{\widetilde{K}}|\!\|_{\widetilde{K}}$ with (3.37) and $\|\!|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})|\!\|_{\omega_{\widetilde{E}}}$ with (3.40). If $\omega_{\widetilde{E}} \cap \widetilde{K} \neq \emptyset$, there holds $\widetilde{K} \subseteq \omega_{\widetilde{E}}$. Therefore,

$$\begin{aligned}
\|\!|w^{(n)}|\!\|^2 \leq & c_4^2\gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 \\
& + 2c_4c_7\gamma_1\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\widetilde{K}\neq\emptyset}} \alpha_{\widetilde{K}} \lambda^{-1/4} \alpha_{\widetilde{E}}^{1/2} \|R_{\widetilde{K}}\|_{\widetilde{K}} \|R_{\widetilde{E}}\|_{\widetilde{E}} \\
& + c_7^2\gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\substack{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}\neq\emptyset}} \lambda^{-1/2} \alpha_{\widetilde{E}}^{1/2} \alpha_{\widetilde{E}'}^{1/2} \|R_{\widetilde{E}}\|_{\widetilde{E}} \|R_{\widetilde{E}'}\|_{\widetilde{E}'}.
\end{aligned}$$

With Young's inequality this yields

$$\begin{aligned}
\|\!|w^{(n)}|\!\|^2 \leq & \, c_4^2\gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}^2} \\
& + c_4^2\gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\widetilde{K}\neq\emptyset}} \Big\{ \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + c_7^2\gamma_2^2 \lambda^{-1/2} \alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \Big\} \\
& + \frac{1}{2} c_7^2\gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \sum_{\substack{\widetilde{E}'\in\widetilde{\mathcal{E}}^{(n)} \\ \omega_{\widetilde{E}}\cap\omega_{\widetilde{E}'}\neq\emptyset}} \Big\{ \lambda^{-1/2} \alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 + \lambda^{-1/2} \alpha_{\widetilde{E}'} \|R_{\widetilde{E}'}\|_{\widetilde{E}'}^2 \Big\}.
\end{aligned}$$

For a triangulation $\widetilde{\mathcal{T}}^{(n)}$, each element $\widetilde{K}$ has three edges $\widetilde{E}_1, \widetilde{E}_2, \widetilde{E}_3 \in \widetilde{\mathcal{E}}^{(n)}$ which satisfy $\omega_{\widetilde{E}_i} \cap \widetilde{K} \neq \emptyset$. On the other hand, a given edge $\widetilde{E}$ belongs to at most two elements $\widetilde{K}_1, \widetilde{K}_2 \in \widetilde{\mathcal{T}}^{(n)}$, i.e. $\omega_{\widetilde{E}} \cap \widetilde{K}_i \neq \emptyset$ which then are neighbors. Accordingly, for an edge $\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)}$ there are at most five edges $E_i \in \widetilde{\mathcal{E}}^{(n)}$ which satisfy $\omega_{\widetilde{E}} \cap \omega_{\widetilde{E}_i} \neq \emptyset$, including $E_i = \widetilde{E}$. Therefore, we obtain

$$\begin{aligned}
|||w^{(n)}|||^2 \leq & c_4^2\gamma_1^2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 \\
& + \max\{c_4^2\gamma_1^2, c_7^2\gamma_2^2\}\Big\{3\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + 2\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2\Big\} \\
& + 5c_7^2\gamma_2^2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
\leq & 7\max\{c_4^2\gamma_1^2, c_7^2\gamma_2^2\}\Big\{ \underbrace{\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2}_{=\left(\eta_h^{(n)}\right)^2} \Big\},
\end{aligned}$$

which proves the second estimate of (3.44). To prove the other inequality, we start with

$$\begin{aligned}
\langle R_h(u_{h,\tau})\,;\, w^{(n)}\rangle &= \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} (R_{\widetilde{K}}\,;\, w^{(n)})_{\widetilde{K}} + \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} (R_{\widetilde{E}}\,;\, w^{(n)})_{\widetilde{E}} \\
&= \gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 (R_{\widetilde{K}}\,;\, \psi_{\widetilde{K}} R_{\widetilde{K}})_{\widetilde{K}} + \gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}} (R_{\widetilde{E}}\,;\, \psi_{\widetilde{E}} R_{\widetilde{E}})_{\widetilde{E}} \\
&\quad + \gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \widetilde{K}\cap\omega_{\widetilde{E}}\neq\emptyset}} \lambda^{-1/2}\alpha_{\widetilde{E}} (R_{\widetilde{K}}\,;\, \psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}}))_{\widetilde{K}}
\end{aligned}$$

since (3.45) and $\psi_{\widetilde{K}} = 0$ on all edges $\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)}$. Using Lemma 3.11, we estimate $(R_{\widetilde{K}}\,;\, \psi_{\widetilde{K}} R_{\widetilde{K}})_{\widetilde{K}}$ and $(R_{\widetilde{E}}\,;\, \psi_{\widetilde{E}} R_{\widetilde{E}})_{\widetilde{E}}$ with (3.36) and (3.38), respectively. We obtain

$$\begin{aligned}
\langle R_h(u_{h,\tau})\,;\, w^{(n)}\rangle &\geq c_3\gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + c_5\gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\quad - \gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \widetilde{K}\cap\omega_{\widetilde{E}}\neq\emptyset}} \lambda^{-1/2}\alpha_{\widetilde{E}} \|R_{\widetilde{K}}\|_{\widetilde{K}} \|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\widetilde{K}},
\end{aligned}$$

where we additionally used Cauchy's inequality for the last term. With (3.39), we obtain $\|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\widetilde{K}} \leq \|\psi_{\widetilde{E}} F_E^{\mathrm{ext}}(R_{\widetilde{E}})\|_{\omega_{\widetilde{E}}} \leq c_6\lambda^{1/4}\alpha_{\widetilde{E}}^{1/2}\|R_{\widetilde{E}}\|_{\widetilde{E}}$. Together with $\alpha_{\widetilde{E}} \leq \alpha_{\widetilde{K}}$ for $\widetilde{K} \cap \omega_{\widetilde{E}} \neq \emptyset$, this implies

$$\begin{aligned}
\langle R_h(u_{h,\tau})\,;\, w^{(n)}\rangle &\geq c_3\gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + c_5\gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\quad - c_6\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \widetilde{K}\cap\omega_{\widetilde{E}}\neq\emptyset}} \lambda^{-1/4}\alpha_{\widetilde{E}}^{1/2}\alpha_{\widetilde{K}} \|R_{\widetilde{K}}\|_{\widetilde{K}} \|R_{\widetilde{E}}\|_{\widetilde{E}} \\
&\geq c_3\gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + c_5\gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\quad - \frac{1}{2} \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \sum_{\substack{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)} \\ \widetilde{K}\cap\omega_{\widetilde{E}}\neq\emptyset}} \Big\{2c_6^2c_5^{-1}\gamma_2\alpha_{\widetilde{K}}^2\|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \frac{1}{2}c_5\gamma_2\lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|_{\widetilde{E}}^2\Big\}.
\end{aligned}$$

There are at most two elements which intersect with a given $\omega_{\widetilde{E}}$, and there are at most three edges such that $\widetilde{K} \cap \omega_{\widetilde{E}}$ is not empty for a given $\widetilde{K}$

$$\begin{aligned}
\langle R_h(u_{h,\tau}) \, ; \, w^{(n)} \rangle &\geq c_3\gamma_1 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + c_5\gamma_2 \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\quad - 3c_6^2c_5^{-1}\gamma_2 \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 - \frac{1}{2}c_5\gamma_2\lambda^{-1/2} \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\geq (c_3\gamma_1 - 3c_6^2c_5^{-1}\gamma_2) \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \frac{1}{2}c_5\gamma_2\lambda^{-1/2} \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \\
&\geq \min\{c_3\gamma_1 - 3c_6^2c_5^{-1}\gamma_2, \frac{1}{2}c_5\gamma_2\} \Big\{ \sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 \\
&\qquad\qquad + \lambda^{-1/2} \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \alpha_{\widetilde{E}} \|R_{\widetilde{E}}\|_{\widetilde{E}}^2 \Big\} \\
&= \min\{c_3\gamma_1 - 3c_6^2c_5^{-1}\gamma_2, \frac{1}{2}c_5\gamma_2\} \Big(\eta_h^{(n)}\Big)^2.
\end{aligned}$$

The choices $\gamma_2 := 2/c_5$ respectively $\gamma_1 := (1 + 6c_6^2/c_5^2)/c_3$ give $\min\{c_3\gamma_1 - 3c_6^2c_5^{-1}\gamma_2, \frac{1}{2}c_5\gamma_2\} = 1$. This proves the first estimate of (3.44). Note that the constants c_3, c_5, c_6 from Lemma (3.11) depend on the shape regularity constant c_S and the transition constant c_T as well as on the polynomial degree of $R_{\widetilde{K}}$ and $R_{\widetilde{E}}$, which is at most 2.

■

3.5 Estimation of the Temporal Residual

After the estimates for the spatial residual, we now want to find respective inequalities for the temporal residual $R_\tau(u_{h,\tau})$. Therefore, we derive upper and lower bounds for the temporal residual on a fixed time interval $[t_{n-1}, t_n]$. For abbreviation, we introduce the notation

$$c_\kappa := 1 + \kappa^2(\max\{c_r, 1\})^2 \geq 2.$$

where κ is the constant from Assumption $(P2)$ and c_r is defined in Assumption $(P4)$.

Lemma 3.13. *For the time interval $(t_{n-1}, t_n]$, there holds*

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} \|\!| R_\tau(u_{h,\tau})(t) |\!\|_*^2 dt \qquad (3.46)\\
&\qquad\qquad \leq \frac{1}{3}c_\kappa\tau_n \Big\{ \|\!| u_h^{(n)} - u_h^{(n-1)} |\!\|^2 + \|\!| \bar{c}^{(n),\theta} \cdot \nabla\big(u_h^{(n)} - u_h^{(n-1)}\big) |\!\|_*^2 \Big\}.
\end{aligned}$$

Here, $u_h^{(n)}$ and $u_h^{(n-1)}$ denote the discrete solution for the time steps t_n and t_{n-1}, respectively. Furthermore $\tau_n = t_n - t_{n-1}$ is the time-step size.

Proof.

1. Claim. *There holds*

$$\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)} - u_{h,\tau} = \Big[\theta - \frac{t - t_{n-1}}{\tau_n}\Big]\big(u_h^{(n)} - u_h^{(n-1)}\big) \qquad (3.47)$$

for $t \in [t_{n-1}, t_n]$.
The function $u_{h,\tau}$ is the affine interpolant of $u_h^{(n)}$ and $u_h^{(n-1)}$ on $t \in [t_{n-1}, t_n]$, i.e.

$$u_{h,\tau} = \frac{t - t_{n-1}}{\tau_n} u_h^{(n)} + \Big(1 - \frac{t - t_{n-1}}{\tau_n}\Big) u_h^{(n-1)}. \tag{3.48}$$

For the right hand side of (3.47), we obtain

$$\Big[\theta - \frac{t - t_{n-1}}{\tau_n}\Big]\big(u_h^{(n)} - u_h^{(n-1)}\big) = \theta u_h^{(n)} - \theta u_h^{(n-1)} - \frac{t - t_{n-1}}{\tau_n} u_h^{(n)} + \frac{t - t_{n-1}}{\tau_n} u_h^{(n-1)}. \tag{3.49}$$

Equation (3.48) and (3.49) now prove (3.47). □

2. Claim. *For the temporal residual, there holds the following representation*

$$R_\tau(u_{h,\tau}) = \Big[\theta - \frac{t - t_{n-1}}{\tau_n}\Big] r_n \tag{3.50}$$

for $t \in [t_{n-1}, t_n]$ *with* $r_n \in H^{-1}$ *defined by the duality pairing*

$$\begin{aligned} \langle r_n \,;\, v\rangle :=& (D^{(n),\theta} \nabla\big(u_h^{(n)} - u_h^{(n-1)}\big) \,;\, \nabla v) + (\bar{c}^{(n),\theta} \cdot \nabla\big(u_h^{(n)} - u_h^{(n-1)}\big) \,;\, v) \\ &+ (r^{(n),\theta}\big(u_h^{(n)} - u_h^{(n-1)}\big) \,;\, v) \quad \forall v \in H_D^1. \end{aligned} \tag{3.51}$$

According to the definition of the temporal residual,

$$\begin{aligned} \langle R_\tau(u_{h,\tau}) \,;\, v\rangle =& (D^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)} - u_{h,\tau}) \,;\, \nabla v) \\ &+ (\bar{c}^{(n),\theta} \cdot \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)} - u_{h,\tau}) \,;\, v) \\ &+ (r^{(n),\theta}(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)} - u_{h,\tau}) \,;\, v) \quad \forall\, v \in H_D^1, \text{ in } (t_{n-1}, t_n], \end{aligned}$$

Equation (3.50) directly follows from the first claim. □

3. Claim. *There holds*

$$\int_{t_{n-1}}^{t_n} \left[\theta - \frac{t - t_{n-1}}{\tau_n}\right]^2 dt = \frac{\tau_n}{3}\big\{\theta^3 + (1-\theta)^3\big\} \text{ for all } \theta \in [1/2, 1]. \tag{3.52}$$

Equation (3.52) is proven by

$$\begin{aligned} \int_{t_{n-1}}^{t_n} \left[\theta - \frac{t - t_{n-1}}{\tau_n}\right]^2 dt &= \int_{t_{n-1}}^{t_n} \left[\theta^2 - 2\theta \frac{t - t_{n-1}}{\tau_n} + \Big(\frac{t - t_{n-1}}{\tau_n}\Big)^2\right] dt \\ &= \tau_n \theta^2 - \frac{\theta}{\tau_n}(t - t_{n-1})^2 \big|_{t_{n-1}}^{t_n} + \frac{1}{3\tau_n^2}(t - t_{n-1})^3 \big|_{t_{n-1}}^{t_n} \\ &= \tau_n \theta^2 - \theta\tau_n + \frac{1}{3}\tau_n \\ &= \frac{\tau_n}{3}(\theta^3 - \theta^3 + 3\theta^2 - 3\theta + 1) \\ &= \frac{\tau_n}{3}\big\{\theta^3 + (1-\theta)^3\big\}. \end{aligned}$$

□

4. Claim. *There hold the following inequalities*

$$\frac{\tau_n}{12} \le \int_{t_{n-1}}^{t_n} \left[\theta - \frac{t - t_{n-1}}{\tau_n}\right]^2 dt \le \frac{\tau_n}{3} \text{ for all } \theta \in [1/2, 1]. \tag{3.53}$$

We use the explicit value of the integral from (3.52) as

$$\frac{d}{d\theta}\Big(\theta^3+(1-\theta)^3\Big) = 3\theta^2 - 3(1-\theta)^2 = 3\theta^2 - 3 + 6\theta - 3\theta^2 = 6(\theta - \frac{1}{2}),$$

which attains its unique minimum for $\theta = 1/2$. Moreover, the unique maximum is attained at $\theta = 1$, on the other boundary of the interval. Evaluation of the right hand side of (3.52) at minimum and maximum prove (3.53).

5. Claim. *The functional r_n satisfies*

$$\|\!|r_n|\!\|_* \leq \sqrt{c_\kappa}\Big\{\|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2\Big\}^{1/2}. \tag{3.54}$$

Estimate (2.17) gives

$$(D^{(n),\theta}\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\nabla w) + (r^{(n),\theta}\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,w) \leq \kappa\max\{c_r,1\}|\!|\!|u_h^{(n)} - u_h^{(n-1)}|\!|\!|\;|\!|\!|w|\!|\!|$$

for all $w \in H^1_D$, since $\big(u_h^{(n)} - u_h^{(n-1)}\big) \in H^1_D$ itself. With the previous equation, the Definition 10 of the norm in the dual space, and Equation (3.51), we obtain

$$\begin{aligned}\|\!|r_n|\!\|_* &\leq \kappa\max\{c_r,1\}\Big\{|\!|\!|u_h^{(n)} - u_h^{(n-1)}|\!|\!| + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*\Big\}\\ &\overset{(2.16)}{\leq} \sqrt{1+\kappa^2(\max\{c_r,1\})^2}\Big\{|\!|\!|u_h^{(n)} - u_h^{(n-1)}|\!|\!|^2 + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2\Big\}^{1/2}.\end{aligned}$$

This proves the claim. □

Finally, we obtain

$$\begin{aligned}\Big\{\int_{t_{n-1}}^{t_n}\|\!|R_\tau(u_{h,\tau})(t)|\!\|_*^2 dt\Big\}^{1/2} &\overset{(3.50)}{=} \Big\{\|\!|r_n|\!\|_*^2\int_{t_{n-1}}^{t_n}\Big[\theta - \frac{t-t_{n-1}}{\tau_n}\Big]^2 dt\Big\}^{1/2}\\ &\overset{(3.53)}{\leq} \|\!|r_n|\!\|_*\Big(\frac{\tau_n}{3}\Big)^{1/2}\\ &\overset{(3.54)}{\leq} \sqrt{\frac{1}{3}c_\kappa\tau_n}\Big\{|\!|\!|u_h^{(n)} - u_h^{(n-1)}|\!|\!|^2\\ &\qquad + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2\Big\}^{1/2},\end{aligned}$$

which concludes the proof. ■

Note, that the expression $\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*$ represents a dual norm and is therefore unpracticable for computations. We will focus on that term below, c.f. Section 3.6.1.

Lemma 3.14. *For every interval $(t_{n-1}, t_n]$, the temporal residual is bounded from below by*

$$\int_{t_{n-1}}^{t_n}\|\!|R_\tau(u_{h,\tau})(t)|\!\|_*^2 dt \geq \frac{1}{24c_\kappa^2}\tau_n\Big\{|\!|\!|u_h^{(n)} - u_h^{(n-1)}|\!|\!|^2 + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2\Big\}. \tag{3.55}$$

Proof.

1. Claim. *For $\delta \in (0,1)$, there is a function $\varphi_{n,\delta} \in H^1_D$ such that*

$$\begin{aligned}|\!|\!|\varphi_{n,\delta}|\!|\!| &= \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*,\\ (\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\varphi_{n,\delta}) &\geq \delta\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2.\end{aligned} \tag{3.56}$$

Let $\phi := \vec{c}^{(n),\theta} \cdot \nabla\big(u_h^{(n)} - u_h^{(n-1)}\big) \in L^2$. Due to the definition of the dual norm $\|\!|\cdot|\!\|_*$

$$\|\!|\phi|\!\|_* = \sup_{v\in H_D^1} \frac{\langle \phi\,;\, v\rangle}{\|\!|v|\!\|} = \sup_{\substack{v\in H_D^1\\ \|\!|v|\!\|=1}} \langle \phi\,;\, v\rangle$$

there is $v \in H_D^1$ with $\|\!|v|\!\| = 1$, such that for $\varepsilon := 1-\delta \in (0,1)$ there holds

$$\langle \phi\,;\, v\rangle \geq \underbrace{(1-\varepsilon)}_{=:\delta} \|\!|\phi|\!\|_*.$$

The definition $\varphi_{n,\delta} := v\|\!|\phi|\!\|_*$ yields $\|\!|\varphi_{n,\delta}|\!\| = \|\!|\phi|\!\|_*$. Moreover, the second inequality of (3.56) is proven by

$$(\phi\,;\,\varphi_{n,\delta}) = \langle \phi\,;\,\varphi_{n,\delta}\rangle = \|\!|\phi|\!\|_* \langle \phi\,;\, v\rangle \geq \delta \|\!|\phi|\!\|_*^2.$$

□

2. Claim. *For*

$$\sigma_{n,\delta} := u_h^{(n)} - u_h^{(n-1)} + \frac{2\delta}{\delta + \kappa^2(\max\{c_r,1\})^2}\varphi_{n,\delta} \tag{3.57}$$

the following estimates hold

$$\begin{aligned} \|\!|\sigma_{n,\delta}|\!\| \leq& \sqrt{2}\Big\{ \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2 \Big\}^{1/2},\\ \langle r_n\,;\,\sigma_{n,\delta}\rangle \geq& \frac{\delta}{\delta + \kappa^2(\max\{c_r,1\})^2}\Big\{ \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + \delta\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2 \Big\} \end{aligned} \tag{3.58}$$

where r_n is defined in (3.51).

For abbreviation we define

$$\gamma := \frac{2\delta}{\delta + \kappa^2(\max\{c_r,1\})^2} \leq \frac{2}{1+\delta^{-1}\kappa^2(\max\{c_r,1\})^2} < 1, \tag{3.59}$$

where we used $\delta \in (0,1)$ and $\kappa \geq 1$. The first inequality in (3.56) then implies

$$\begin{aligned} \|\!|\sigma_{n,\delta}|\!\| \leq& \max\{1,\gamma\}\Big\{ \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\| + \|\!|\varphi_{n,\delta}|\!\| \Big\}\\ =& \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\| + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*.\\ \leq& \sqrt{2}\Big\{ \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + \|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!\|_*^2 \Big\}^{1/2} \end{aligned}$$

With $(u_h^{(n)} - u_h^{(n-1)}), \varphi_{n,\delta} \in H_D^1$ there holds $\sigma_{n,\delta} \in H_D^1$. Thus, we can use ellipticity (2.12) and Estimate (2.17) to obtain

$$\begin{aligned} \langle r_n\,;\,\sigma_{n,\delta}\rangle =& (D^{(n),\theta}\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big))\\ &+ (\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,u_h^{(n)} - u_h^{(n-1)})\\ &+ (r^{(n),\theta}\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,u_h^{(n)} - u_h^{(n-1)}) + \gamma(D^{(n),\theta}\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\nabla\varphi_{n,\delta})\\ &+ \gamma(\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\varphi_{n,\delta}) + \gamma(r^{(n),\theta}\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\varphi_{n,\delta})\\ \geq& \|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + \gamma(\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)\,;\,\varphi_{n,\delta})\\ &- \gamma\kappa\max\{c_r,1\}\|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|\,\|\!|\varphi_{n,\delta}|\!\|. \end{aligned}$$

Furthermore, Equation (3.56) and Young's inequality implies

$$\begin{aligned}\langle r_n\,;\,\sigma_{n,\delta}\rangle \geq & \|\!|u_h^{(n)}-u_h^{(n-1)}|\!\|^2+\gamma\delta\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)}-u_h^{(n-1)}\big)|\!\|_*^2\\ &-\gamma\kappa\max\{c_r,1\}\|\!|u_h^{(n)}-u_h^{(n-1)}|\!\|\,\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)}-u_h^{(n-1)}\big)|\!\|_*\\ \geq&\left\{1-\frac{1}{2}\gamma\delta^{-1}\kappa^2(\max\{c_r,1\})^2\right\}\|\!|u_h^{(n)}-u_h^{(n-1)}|\!\|^2\\ &+\frac{1}{2}\gamma\delta\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)}-u_h^{(n-1)}\big)|\!\|_*^2,\end{aligned}$$

Now, (3.59) and

$$1-\frac{1}{2}\gamma\delta^{-1}\kappa^2(\max\{c_r,1\})^2=1-\frac{\kappa^2(\max\{c_r,1\})^2}{\delta+\kappa^2(\max\{c_r,1\})^2}=\frac{\delta}{\delta+\kappa^2(\max\{c_r,1\})^2}$$

prove the second estimate in (3.58). □

Finally, with the definition

$$z_{n,\delta}:=\left[\theta-\frac{t-t_{n-1}}{\tau_n}\right]\sigma_{n,\delta},\tag{3.60}$$

we use the representation of $R_\tau(u_{h,\tau})$ defined in (3.50) to obtain

$$\begin{aligned}\left\{\int_{t_{n-1}}^{t_n}\|\!|R_\tau(u_{h,\tau}(t))|\!\|_*^2dt\right\}^{1/2}\geq&\left\{\int_{t_{n-1}}^{t_n}\frac{\langle R_\tau(u_{h,\tau}(t))\,;\,z_{n,\delta}(t)\rangle^2}{\|\!|z_{n,\delta}(t)|\!\|^2}dt\right\}^{1/2}\\ =&\frac{\langle r_n\,;\,\sigma_{n,\delta}\rangle}{\|\!|\sigma_{n,\delta}|\!\|}\left\{\int_{t_{n-1}}^{t_n}\left[\theta-\frac{t-t_{n-1}}{\tau_n}\right]^2dt\right\}^{1/2}.\end{aligned}$$

The integral is estimated by the lower bound $\tau_n^{1/2}/\sqrt{12}$ in (3.53), and with (3.58) in the limit $\delta\to 1$ the previous equation yields

$$\begin{aligned}&\left\{\int_{t_{n-1}}^{t_n}\|\!|R_\tau(u_{h,\tau}(t))|\!\|_*^2dt\right\}^{1/2}\\ &\quad\geq\frac{\tau_n^{1/2}}{\sqrt{24}[1+\kappa^2(\max\{c_r,1\})^2]}\left\{\|\!|u_h^{(n)}-u_h^{(n-1)}|\!\|^2+\|\!|\vec{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)}-u_h^{(n-1)}\big)|\!\|_*^2\right\}^{1/2}.\end{aligned}$$

Since $c_\kappa=1+\kappa^2(\max\{c_r,1\})^2$, this proves (3.55).

■

3.6 A posteriori Error Estimates

After the preparations of the previous sections, we can put the pieces together. We present upper and lower bound of the error. Note, that the following upper bound is global with respect to time and space, whereas the lower bound is local in time but global in space. For abbreviation, we define the **temporal data error estimator** and the **Neumann error estimator** by

$$\varrho_\tau^{(n)}(u_{h,\tau}):=\sup_{v\in H_D^1\setminus\{0\}}\frac{((D^{(n),\theta}-D)\nabla u_{h,\tau}\,;\,\nabla v)}{\|\!|v|\!\|}+\|\!|(\vec{c}^{(n),\theta}-\vec{c})\cdot\nabla u_{h,\tau}+(r^{(n),\theta}-r)u_{h,\tau}|\!\|_*\tag{3.61}$$

and

$$\eta_N^{(n)} := \lambda^{-1/4}\Big\{ \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}_{\Gamma_N}} \alpha_{\widetilde{E}}\Big[\theta^2\|g(t_n)-g_h^{(n)}\|_{\widetilde{E}}^2 + (1-\theta)^2\|g(t_{n-1})-g_h^{(n-1)}\|_{\widetilde{E}}^2\Big]\Big\}^{1/2}, \tag{3.62}$$

respectively, for all time steps n with $n = 1, \dots, N$.

Theorem 3.15 (Upper Bound). *The error between the solution u of the weak form (2.26) and the discrete solution $u_{h,\tau}$ of (2.29) and (2.31) is bounded from above by*

$$\begin{aligned}
&\|u-u_{h,\tau}\|_{X(0,T)}^2 \\
&\quad\lesssim \|u_0-\Pi_0 u_0\|^2 \\
&\qquad + \sum_{n=1}^N \tau_n\Big[\Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2 + |\!|\!|u_h^{(n)}-u_h^{(n-1)}|\!|\!|^2 + |\!|\!|\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|\!|\!|_*^2 \\
&\qquad\qquad + \Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big] + \sum_{n=1}^N\int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt \\
&\qquad + \|f-f_{h,\tau}\|_{L^2(0,T;H^{-1})}^2
\end{aligned} \tag{3.63}$$

with a constant depending on the shape regularity constant c_S, the transition constant c_T, the constants κ from (P2) and c_r from (P4), the Poincaré constant c_P, and on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$.

Proof. With (3.11) and the decomposition of the residual $R(u_{h,\tau})$ into $R_\tau(u_{h,\tau})$, $R_h(u_{h,\tau})$, and $R_D(u_{h,\tau})$ according to Lemma 3.7, we obtain

$$\begin{aligned}
\|u-u_{h,\tau}\|_{X(0,T)}^2 &\overset{(3.11)}{\leq} 2c_\kappa\|u_0-\Pi_0u_0\|^2 + 2(2+\kappa^2(\max\{c_r,1\})^2)\|R(u_{h,\tau})\|_{L^2(0,T,H_D^{-1})}^2 \\
&\overset{\text{Def.11}}{\lesssim} \|u_0-\Pi_0u_0\|^2 + \int_0^T |\!|\!|R(u_{h,\tau})|\!|\!|_*^2 dt \\
&\lesssim \|u_0-\Pi_0u_0\|^2 + \int_0^T |\!|\!|R_h(u_{h,\tau})|\!|\!|_*^2 dt \\
&\qquad + \int_0^T |\!|\!|R_\tau(u_{h,\tau})|\!|\!|_*^2 dt + \int_0^T |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2 dt.
\end{aligned}$$

The spatial residual and the temporal residual are estimated by (3.31) and (3.46), respectively

$$\begin{aligned}
\|u-u_{h,\tau}\|_{X(0,T)}^2 &\lesssim \|u_0-\Pi_0u_0\|^2 + \sum_{n=1}^N \tau_n\Big[\Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2\Big] \\
&\qquad + \int_0^T |\!|\!|R_\tau(u_{h,\tau})|\!|\!|_*^2 dt + \int_0^T |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2 dt \\
&\lesssim \|u_0-\Pi_0u_0\|^2 + \sum_{n=1}^N \tau_n\Big[\Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2\Big] \\
&\qquad + \sum_{n=1}^N \tau_n\Big[|\!|\!|u_h^{(n)}-u_h^{(n-1)}|\!|\!|^2 + |\!|\!|\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|\!|\!|_*^2\Big] \\
&\qquad + \int_0^T |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2 dt.
\end{aligned}$$

It thus only remains to estimate the data residual term. This is achieved within several steps.

1. Claim. *For $v \in H_0^1$ and $t \in (t_{n-1}, t_n]$ with $1 \leq n \leq N$, we have the estimate*

$$(g - g_{h,\tau}\,;\, v)_{\Gamma_N} \lesssim \left\{ \left(\eta_N^{(n)}\right)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds \right\}^{1/2} |\!|\!| v |\!|\!| \tag{3.64}$$

with a constant that depends on the shape regularity constant c_S and the transition constant c_T.
A straightforward consideration shows

$$g(t) = \theta g(t_n) + (1-\theta) g(t_{n-1}) + \int_{t_{n-1}}^{t} \dot{g}(s) ds - \theta \int_{t_{n-1}}^{t_n} \dot{g}(s) ds.$$

Recalling that $g_{h,\tau} = \theta g_h^{(n)} + (1-\theta) g_h^{(n)}$ for $t \in (t_{n-1}, t_n]$ implies

$$\begin{aligned}(g - g_{h,\tau}\,;\, v)_{\Gamma_N} =& \theta (g(t_n) - g_h^{(n)}\,;\, v)_{\Gamma_N} + (1-\theta)(g(t_{n-1}) - g_h^{(n-1)}\,;\, v)_{\Gamma_N} \\ &+ \left(\int_{t_{n-1}}^{t} \dot{g}(s) ds - \theta \int_{t_{n-1}}^{t_n} \dot{g}(s) ds \,;\, v \right)_{\Gamma_N}.\end{aligned}$$

Let Π_{h,Γ_N} be the L^2-projection onto the space of Γ_N-traces of functions in $\mathcal{S}_D^1(\widetilde{\mathcal{T}}^{(n)})$. Since $g(t_n) - g_h^{(n)}$ is in the orthogonal complement of this space we have that

$$\begin{aligned}(g(t_n) - g_h^{(n)}\,;\, v)_{\Gamma_N} =& (g(t_n) - g_h^{(n)}\,;\, v - \Pi_{h,\Gamma_N} v)_{\Gamma_N} \\ \leq& \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}_{\Gamma_N}} \|g(t_n) - g_h^{(n)}\|_{\widetilde{E}} \|v - \Pi_{h,\Gamma_N} v\|_{\widetilde{E}},\end{aligned}$$

where $\widetilde{\mathcal{E}}_{\Gamma_N}$ denotes the set of all Neumann edges of $\widetilde{\mathcal{T}}^{(n)}$. The projection Π_{h,Γ_N} is the best approximation operator, i.e.

$$\|v - \Pi_{h,\Gamma_N} v\|_{\widetilde{E}} \leq \|v - I_h^{(n)} v\|_{\widetilde{E}},$$

where $I_h^{(n)}$ denotes the Clément operator for the triangulation $\widetilde{\mathcal{T}}^{(n)}$, c.f Definition 24. With (3.26), we obtain

$$\begin{aligned}(g(t_n) - g_h^{(n)}\,;\, v)_{\Gamma_N} \lesssim& \lambda^{-1/4} \sum_{\substack{\widetilde{E} \in \widetilde{\mathcal{E}}_{\Gamma_N} \\ K \in \mathcal{T}^{(n)}, K \cap \widetilde{E} \neq \emptyset}} \alpha_{\widetilde{E}}^{1/2} \|g(t_n) - g_h^{(n)}\|_{\widetilde{E}} |\!|\!| v |\!|\!|_{\omega_K} \\ \lesssim& \lambda^{-1/4} \left\{ \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}_{\Gamma_N}} \alpha_{\widetilde{E}} \|g(t_n) - g_h^{(n)}\|_{\widetilde{E}}^2 \right\}^{1/2} \left\{ \sum_{K \in \mathcal{T}^{(n)}} |\!|\!| v |\!|\!|_{\omega_K}^2 \right\}^{1/2}.\end{aligned}$$

Both estimates in the previous two inequalities depend on the shape regularity constant c_S and the transition constant c_T. Moreover, we have the estimate

$$\left\{ \sum_{K \in \mathcal{T}^{(n)}} |\!|\!| v |\!|\!|_{\omega_K}^2 \right\}^{1/2} \lesssim \left\{ \sum_{K \in \mathcal{T}^{(n)}} |\!|\!| v |\!|\!|_K^2 \right\}^{1/2} = |\!|\!| v |\!|\!|.$$

Thus,

$$\begin{aligned}\theta(g(t_n) - g_h^{(n)}\,;\,v)_{\Gamma_N} &+ (1-\theta)(g(t_{n-1}) - g_h^{(n-1)}\,;\,v)_{\Gamma_N}\\ &\lesssim \theta\lambda^{-1/4}\Big\{\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}_{\Gamma_N}} \alpha_{\widetilde{E}}\|g(t_n) - g_h^{(n)}\|_{\widetilde{E}}^2\Big\}^{1/2}|\!|\!|v|\!|\!|\\ &\quad + (1-\theta)\lambda^{-1/4}\Big\{\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}_{\Gamma_N}} \alpha_{\widetilde{E}}\|g(t_{n-1}) - g_h^{(n-1)}\|_{\widetilde{E}}^2\Big\}^{1/2}|\!|\!|v|\!|\!|\\ &\lesssim \eta_N^{(n)}|\!|\!|v|\!|\!|.\end{aligned}$$

With the continuous trace operator $\gamma \in L(H^1(\Omega), H^{1/2}(\Gamma))$, we obtain

$$\|v\|_{\Gamma_N} \le \|v\|_{H^{1/2}(\Gamma)} \lesssim |\!|\!|v|\!|\!|.$$

Hence, we have

$$\begin{aligned}(g - g_{h,\tau}\,;\,v)_{\Gamma_N} \lesssim& \eta_N^{(n)}|\!|\!|v|\!|\!| + \Big(\int_{t_{n-1}}^{t}\dot g(s)ds - \theta\int_{t_{n-1}}^{t_n}\dot g(s)ds\,;\,v\Big)_{\Gamma_N}\\ \le& \eta_N^{(n)}|\!|\!|v|\!|\!| + \Big[\int_{t_{n-1}}^{t}\|\dot g(s)\|_{\Gamma_N}ds + \theta\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}ds\Big]|\!|\!|v|\!|\!|\\ \le& \eta_N^{(n)}|\!|\!|v|\!|\!| + \Big[\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}ds + \theta\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}ds\Big]|\!|\!|v|\!|\!|\\ \lesssim& \Big\{\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}^2ds\Big\}^{1/2}|\!|\!|v|\!|\!|,\end{aligned}$$

where we used Cauchy's inequality twice for the last estimate. This proves the claim. □

2. Claim. *The integral of the dual norm of the data residual $R_D(u_{h,\tau})$ can be estimated by*

$$\begin{aligned}\int_0^T |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2dt \lesssim& \|f - f_{h,\tau}\|_{L^2(0,T;H^{-1})}^2 + \sum_{n=1}^N\tau_n\Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}^2ds\Big]\\ &+ \sum_{n=1}^N\int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt.\end{aligned} \tag{3.65}$$

Recalling Definition 18 and Equation (3.64) imply

$$\begin{aligned}\langle R_D(u_{h,\tau})\,;\,v\rangle =& (f - f_{h,\tau}\,;\,v) + (g - g_{h,\tau}\,;\,v)_{\Gamma_N}\\ &+ ((D^{(n),\theta} - D)\nabla u_{h,\tau}\,;\,\nabla v)\\ &+ ((\vec c^{(n),\theta} - \vec c)\cdot\nabla u_{h,\tau} + (r^{(n),\theta} - r)u_{h,\tau}\,;\,v)\\ \lesssim& (f - f_{h,\tau}\,;\,v) + \Big\{\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|_{\Gamma_N}^2ds\Big\}^{1/2}|\!|\!|v|\!|\!|\\ &+ ((D^{(n),\theta} - D)\nabla u_{h,\tau}\,;\,\nabla v)\\ &+ ((\vec c^{(n),\theta} - \vec c)\cdot\nabla u_{h,\tau} + (r^{(n),\theta} - r)u_{h,\tau}\,;\,v).\end{aligned}$$

With (3.61), we obtain

$$|||R_D(u_{h,\tau})|||_*^2 \lesssim |||f - f_{h,\tau}|||_*^2 + \left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds + \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2,$$

and therefore

$$\begin{aligned}\int_0^T |||R_D(u_{h,\tau})|||_*^2 dt \lesssim & \|f - f_{h,\tau}\|_{L^2(0,T;H^{-1})}^2 + \sum_{n=1}^N \tau_n \Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big] \\ & + \sum_{n=1}^N \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt,\end{aligned}$$

which proves the claim. □

Finally, with (3.65) we conclude

$$\begin{aligned}\|u - u_{h,\tau}\|_{X(0,T)}^2 \lesssim & \|u_0 - \Pi_0 u_0\|^2 + \sum_{n=1}^N \tau_n \Big[\left(\eta_h^{(n)}\right)^2 + \left(\Theta_h^{(n)}\right)^2\Big] \\ & + \sum_{n=1}^N \tau_n \Big[|||u_h^{(n)} - u_h^{(n-1)}|||^2 + |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})|||_*^2\Big] \\ & + \|f - f_{h,\tau}\|_{L^2(0,T;H^{-1})}^2 + \sum_{n=1}^N \tau_n \Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big] \\ & + \sum_{n=1}^N \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt,\end{aligned}$$

which proves the theorem. ■

For the respective lower bound we again restrict to data D, $\vec{c}$, r that are constant with respect to space. Thus, all the data error estimators $\Theta_h^{(n)}$ are trivial.

Theorem 3.16 (Lower Bound). *Let the data D, $\vec{c}$, r be spatially constant. For each interval $(t_{n-1}, t_n]$ with $1 \le n \le N$ the error is bounded from below by*

$$\begin{aligned}\tau_n \Big\{ \left(\eta_h^{(n)}\right)^2 + |||u_h^{(n)} - u_h^{(n-1)}|||^2 + |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})|||_*^2 \Big\} & \\ \lesssim \|u - u_{h,\tau}\|_{X(t_{n-1},t_n)}^2 + \|f - f_{h,\tau}\|_{L^2(t_{n-1},t_n;H^{-1})}^2 & \\ + \tau_n \Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big] + \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt & \end{aligned} \tag{3.66}$$

with a constant that depends only on the shape regularity constant c_S, the transition constant c_T, and the constants κ from $(P2)$ and c_r from $(P4)$.

Proof.

1. Claim. *For the last two terms of the left hand side of (3.66), there holds*

$$\begin{aligned}\tau_n \Big\{ |||u_h^{(n)} - u_h^{(n-1)}|||^2 + |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})|||_*^2 \Big\} & \\ \le 24 c_\kappa \Big\{ c_\kappa \|u - u_{h,\tau}\|_{X(t_{n-1},t_n)}^2 + c_{h,u}^2 \tau_n \left(\eta_h^{(n)}\right)^2 + c_D \|f - f_{h,\tau}\|_{L^2(t_{n-1},t_n;H^{-1})}^2 & \\ + c_D \tau_n \Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big] + \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt \Big\} & \end{aligned} \tag{3.67}$$

with constants $c_{h,u}$,c_D determined below.
The lower bound (3.55) for the temporal residual implies

$$\tau_n\Big\{\|\!|u_h^{(n)} - u_h^{(n-1)}|\!\|^2 + |\!|\!|\bar{c}^{(n),\theta}\cdot\nabla\big(u_h^{(n)} - u_h^{(n-1)}\big)|\!|\!|_*^2\Big\} \leq 24c_\kappa \int_{t_{n-1}}^{t_n} |\!|\!|R_\tau(u_{h,\tau})|\!|\!|_*^2 dt.$$

We decompose the temporal residual $R_\tau(u_{h,\tau}) = R(u_{h,\tau}) - R_h(u_{h,\tau}) - R_D(u_{h,\tau})$, according to Lemma 3.7

$$\int_{t_{n-1}}^{t_n} |\!|\!|R_\tau(u_{h,\tau})|\!|\!|_*^2 dt \leq \int_{t_{n-1}}^{t_n} |\!|\!|R(u_{h,\tau})|\!|\!|_*^2 dt + \int_{t_{n-1}}^{t_n} |\!|\!|R_h(u_{h,\tau})|\!|\!|_*^2 dt + \int_{t_{n-1}}^{t_n} |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2 dt.$$

Now, we estimate each term in the previous expression separately. First, note that Lemma 3.5 applies to each time interval (t_{n-1}, t_n) and this yields

$$\int_{t_{n-1}}^{t_n} |\!|\!|R(u_{h,\tau})|\!|\!|_*^2 dt \leq c_\kappa \|u - u_{h,\tau}\|_{X(t_{n-1},t_n)}^2.$$

Since the spatial residual is constant on each time interval, there holds

$$\int_{t_n-1}^{t_n} |\!|\!|R_h(u_{h,\tau}(.,t))|\!|\!|_*^2 dt = |\!|\!|R_h(u_{h,\tau})|\!|\!|_*^2 \Big\{ \int_{t_n-1}^{t_n} dt \Big\} \leq c_{h,u}^2 \tau_n \Big(\eta_h^{(n)}\Big)^2,$$

with the constant $c_{h,u}$ defined in (3.31). For the data residual $R_D(u_{h,\tau})$, we use Estimate (3.65) which implies

$$\begin{aligned}\int_{t_{n-1}}^{t_n} |\!|\!|R_D(u_{h,\tau})|\!|\!|_*^2 dt \leq & c_D\Big\{\|f - f_{h,\tau}\|_{L^2(t_{n-1},t_n;H^{-1})}^2 + \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt \\ & + \tau_n\Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|_{\Gamma_N}^2 ds\Big]\Big\},\end{aligned}$$

where c_D denotes the constant from this estimate. This proves (3.67). □

In the second part of this proof, we estimate the term $\tau_n\Big(\eta_h^{(n)}\Big)^2$, which shows up on the left hand side of (3.66) as well as on the right hand side of (3.67). We start with an auxiliary consideration:

2. Claim. *For $\alpha \in \mathbb{R}_+$ the following equation holds*

$$\int_{t_{n-1}}^{t_n} (\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha \Big[\theta - \frac{t-t_{n-1}}{\tau_n}\Big] dt = \Big(\theta - \frac{\alpha+1}{\alpha+2}\Big)\tau_n. \tag{3.68}$$

An integration by parts of the left hand side gives

$$\begin{aligned}&\theta \int_{t_{n-1}}^{t_n} (\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha dt - \int_{t_{n-1}}^{t_n} (\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha \Big[\frac{t-t_{n-1}}{\tau_n}\Big] dt \\ &= \theta\tau_n - \tau_n\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^{\alpha+1}\Big|_{t_{n-1}}^{t_n}\Big[\frac{t-t_{n-1}}{\tau_n}\Big]\Big|_{t_{n-1}}^{t_n} + \int_{t_{n-1}}^{t_n}\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^{\alpha+1} dt \\ &= \Big(\theta - 1 + \frac{1}{\alpha+2}\Big)\tau_n \\ &= \Big(\theta - \frac{\alpha+1}{\alpha+2}\Big)\tau_n.\end{aligned}$$

This proves the claim. □

3. Claim. *For $\alpha \in \mathbb{R}_+$ and the spatial Verfürth-type error estimator $\eta_h^{(n)}$, there holds*

$$\begin{aligned}\tau_n\Big(\eta_h^{(n)}\Big)^2 \leq & c_8\left|\theta - \frac{\alpha+1}{\alpha+2}\right|\tau_n\Big(\eta_h^{(n)}\Big)^2 \\ & + c_{10}\left[\sqrt{\alpha+1} + \left|\theta - \frac{\alpha+1}{\alpha+2}\right|\right]\tau_n^{1/2}\eta_h^{(n)}\times \\ & \times\Bigg\{\|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)} + \|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\ & + \tau_n\Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big] + \int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\Bigg\}^{1/2}\end{aligned} \tag{3.69}$$

with constants c_8 and c_{10} that depend only on the shape regularity constant c_S, the transition constant c_T, and on κ and c_r.

Recall the function $w^{(n)}$ defined in (3.45)

$$w^{(n)} = \gamma_1\sum_{\widetilde K\in\widetilde{\mathcal T}^{(n)}}\alpha^2_{\widetilde K}\psi_{\widetilde K}R_{\widetilde K} + \gamma_2\sum_{\widetilde E\in\widetilde{\mathcal E}^{(n)}}\lambda^{-1/2}\alpha_{\widetilde E}\psi_{\widetilde E}F_{\widetilde E}^{\mathrm{ext}}(R_{\widetilde E}),$$

where $\gamma_2 = 2/c_5$ and $\gamma_1 = (1+6c_6^2/c_5^2)/c_3$ are the constants defined in the proof of Lemma 3.12. Moreover, $\psi_{\widetilde K}$, $\psi_{\widetilde E}$, $R_{\widetilde K}$, $R_{\widetilde E}$ are the bubble function of an element, the bubble function of an edge, the element residual, and the edge residual, respectively. All quantities of $w^{(n)}$ are polynomials, hence $(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha w^{(n)} \in L^2(t_{n-1},t_n,H^1_D)$ for $\alpha\in\mathbb{R}_+$. With

$$\int_{t_{n-1}}^{t_n}(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha dt = \frac{(t-t_{n-1})^{\alpha+1}}{\tau_n^\alpha}\Big|_{t_{n-1}}^{t_n} = \tau_n$$

and with the lower bound (3.44) of the spatial residual, we obtain

$$\begin{aligned}\tau_n\Big(\eta_h^{(n)}\Big)^2 &= \Big(\eta_h^{(n)}\Big)^2\int_{t_{n-1}}^{t_n}(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha dt \\ &\leq \int_{t_{n-1}}^{t_n}(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha\langle R_h(u_{h,\tau})\,;\,w^{(n)}\rangle dt \\ &\overset{(3.20)}{=} \int_{t_{n-1}}^{t_n}(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha\langle R(u_{h,\tau}) - R_\tau(u_{h,\tau}) - R_D(u_{h,\tau})\,;\,w^{(n)}\rangle dt.\end{aligned}$$

Again, we estimate the terms on the right hand side separately. We start with the residual $R(u_{h,\tau})$

$$\begin{aligned}&\int_{t_{n-1}}^{t_n}(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha\langle R(u_{h,\tau})\,;\,w^{(n)}\rangle dt \\ &\qquad\overset{(3.9)}{\leq} \sqrt{c_\kappa}\|u-u_{h,\tau}\|_{X(t_{n-1},t_n)}\left\|(\alpha+1)\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^\alpha w^{(n)}\right\|_{L^2(t_{n-1},t_n,H^1_D)} \\ &\qquad= \sqrt{c_\kappa}\|u-u_{h,\tau}\|_{X(t_{n-1},t_n)}\Big\{\int_{t_{n-1}}^{t_n}(\alpha+1)^2\Big(\frac{t-t_{n-1}}{\tau_n}\Big)^{2\alpha}dt\Big\}^{1/2}|\!|\!|w^{(n)}|\!|\!|,\end{aligned}$$

since $w^{(n)}$ is constant with respect to time on $(t_{n-1},t_n]$. Due to (3.44), the norm $|\!|\!|w^{(n)}|\!|\!|$ is bounded by

$$|\!|\!|w^{(n)}|\!|\!| \leq c_{h,\ell}\eta_h^{(n)}.$$

Moreover, a straightforward consideration proves

$$\left\{ \int_{t_{n-1}}^{t_n} (\alpha+1)^2 \left(\frac{t-t_{n-1}}{\tau_n} \right)^{2\alpha} dt \right\}^{1/2} = \left\{ \frac{(\alpha+1)^2}{2\alpha+1} \right\}^{1/2} \tau_n^{1/2} \overset{\alpha \in \mathbb{R}_+}{\leq} \sqrt{\alpha+1}\, \tau_n^{1/2}.$$

Altogether, we obtain

$$\int_{t_{n-1}}^{t_n} (\alpha+1) \left(\frac{t-t_{n-1}}{\tau_n} \right)^{\alpha} \langle R(u_{h,\tau}) \, ; \, w^{(n)} \rangle dt \leq \sqrt{c_\kappa} c_{h,\ell} \sqrt{\alpha+1}\, \tau_n^{1/2} \eta_h^{(n)} \| u - u_{h,\tau} \|_{X(t_{n-1},t_n)}.$$

Furthermore, with the representation of the temporal residual from (3.50) and Equation (3.68),

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} (\alpha+1) \left(\frac{t-t_{n-1}}{\tau_n} \right)^{\alpha} \langle R_\tau(u_{h,\tau}) \, ; \, w^{(n)} \rangle dt \\
&\quad = \langle r_n \, ; \, w^{(n)} \rangle \int_{t_{n-1}}^{t_n} \Big[\theta - \frac{t-t_{n-1}}{\tau_n} \Big] (\alpha+1) \left(\frac{t-t_{n-1}}{\tau_n} \right)^{\alpha} dt \\
&\quad = \left(\theta - \frac{\alpha+1}{\alpha+2} \right) \tau_n \langle r_n \, ; \, w^{(n)} \rangle \\
&\quad \leq \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n |\!|\!| r_n |\!|\!|_* |\!|\!| w^{(n)} |\!|\!|.
\end{aligned}$$

The dual norm $|\!|\!| r_n |\!|\!|_*$ can be estimated according to (3.54), and thus

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} (\alpha+1) \left(\frac{t-t_{n-1}}{\tau_n} \right)^{\alpha} \langle R_\tau(u_{h,\tau}) \, ; \, w^{(n)} \rangle dt \\
&\quad \leq \sqrt{c_\kappa} \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n |\!|\!| w^{(n)} |\!|\!| \left\{ |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + |\!|\!| \vec{c}^{(n),\theta} \cdot \nabla \big(u_h^{(n)} - u_h^{(n-1)} \big) |\!|\!|_*^2 \right\}^{1/2} \\
&\quad \leq \sqrt{c_\kappa} c_{h,\ell} \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n \eta_h^{(n)} \left\{ |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + |\!|\!| \vec{c}^{(n),\theta} \cdot \nabla \big(u_h^{(n)} - u_h^{(n-1)} \big) |\!|\!|_*^2 \right\}^{1/2},
\end{aligned}$$

where we used the estimate for $|\!|\!| w^{(n)} |\!|\!|$ from above. The right hand side of the previous estimate may be further estimated by (3.67), implying

$$\begin{aligned}
&\sqrt{c_\kappa} c_{h,\ell} \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n \eta_h^{(n)} \left\{ |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + |\!|\!| \vec{c}^{(n),\theta} \cdot \nabla \big(u_h^{(n)} - u_h^{(n-1)} \big) |\!|\!|_*^2 \right\}^{1/2} \\
&\leq \sqrt{c_\kappa} c_{h,\ell} \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n^{1/2} \eta_h^{(n)} \times \\
&\qquad \times \sqrt{24} \sqrt{c_\kappa} \Bigg\{ c_\kappa \| u - u_{h,\tau} \|_{X(t_{n-1},t_n)}^2 + c_{h,u}^2 \tau_n \left(\eta_h^{(n)} \right)^2 + c_D \| f - f_{h,\tau} \|_{L^2(t_{n-1},t_n;H^{-1})}^2 \\
&\qquad\qquad + c_D \tau_n \Big[\left(\eta_N^{(n)} \right)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \| \dot{g}(s) \|_{\Gamma_N}^2 ds \Big] + \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau}) \big]^2 dt \Bigg\}^{1/2} \\
&\leq c_8 \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n \left(\eta_h^{(n)} \right)^2 \\
&\quad + c_9 \left| \theta - \frac{\alpha+1}{\alpha+2} \right| \tau_n^{1/2} \eta_h^{(n)} \Bigg\{ \| u - u_{h,\tau} \|_{X(t_{n-1},t_n)}^2 + \| f - f_{h,\tau} \|_{L^2(t_{n-1},t_n;H^{-1})}^2 \\
&\qquad\qquad + \tau_n \Big[\left(\eta_N^{(n)} \right)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \| \dot{g}(s) \|_{\Gamma_N}^2 ds \Big] \\
&\qquad\qquad + \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau}) \big]^2 dt \Bigg\}^{1/2}
\end{aligned}$$

with c_8 and c_9 constants depending on κ, c_r, $c_{h,\ell}$ and c_D, and c_8 additionally depending on $c_{h,u}$. In conclusion, we obtain an estimate for the temporal residual

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} (\alpha+1)\left(\frac{t-t_{n-1}}{\tau_n}\right)^{\alpha} \langle R_\tau(u_{h,\tau})\,;\, w^{(n)}\rangle dt \\
&\qquad \le c_8\left|\theta - \frac{\alpha+1}{\alpha+2}\right| \tau_n \left(\eta_h^{(n)}\right)^2 \\
&\qquad\quad + c_9\left|\theta - \frac{\alpha+1}{\alpha+2}\right| \tau_n^{1/2}\eta_h^{(n)} \Bigg\{ \|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)} + \|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad\qquad\qquad + \tau_n\Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot g(s)\|^2_{\Gamma_N} ds\Big] \\
&\qquad\qquad\qquad + \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt \Bigg\}^{1/2}.
\end{aligned}$$

To estimate the data residual $R_D(u_{h,\tau})$, we use Cauchy's inequality and (3.65)

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} (\alpha+1)\left(\frac{t-t_{n-1}}{\tau_n}\right)^{\alpha} \langle R_D(u_{h,\tau})\,;\, w^{(n)}\rangle dt \\
&\qquad \le |\!|\!| w^{(n)} |\!|\!| \left\{ \int_{t_{n-1}}^{t_n} |\!|\!| R_D |\!|\!|_*^2 dt \right\}^{1/2} \left\{ \int_{t_{n-1}}^{t_n} (\alpha+1)^2 \left(\frac{t-t_{n-1}}{\tau_n}\right)^{2\alpha} dt \right\}^{1/2} \\
&\qquad \le |\!|\!| w^{(n)} |\!|\!| \left\{ \int_{t_{n-1}}^{t_n} (\alpha+1)^2 \left(\frac{t-t_{n-1}}{\tau_n}\right)^{2\alpha} dt \right\}^{1/2} \times \\
&\qquad\quad \times \Big\{ \|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad\qquad + \tau_n\Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot g(s)\|^2_{\Gamma_N} ds\Big] + \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt \Big\}^{1/2},
\end{aligned}$$

For the term $|\!|\!| w^{(n)} |\!|\!|$ and the time integral, we employ the estimates from above, which provide

$$\begin{aligned}
&\int_{t_{n-1}}^{t_n} (\alpha+1)\left(\frac{t-t_{n-1}}{\tau_n}\right)^{\alpha} \langle R_D(u_{h,\tau})\,;\, w^{(n)}\rangle dt \\
&\qquad \le c_{h,\ell}\sqrt{\alpha+1}\,\tau_n^{1/2}\eta_h^{(n)} \\
&\qquad\quad \times \Big\{ \|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad\qquad + \tau_n\Big[\left(\eta_N^{(n)}\right)^2 + (1+\theta)^2\tau_n \int_{t_{n-1}}^{t_n} \|\dot g(s)\|^2_{\Gamma_N} ds\Big] + \int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt \Big\}^{1/2}.
\end{aligned}$$

To obtain (3.69), we now use the estimates for the residual, the temporal residual and the data residual, respectively,

$$\begin{aligned}
&\tau_n\Big(\eta_h^{(n)}\Big)^2 \\
&\quad\le \int_{t_{n-1}}^{t_n}(\alpha+1)\bigg(\frac{t-t_{n-1}}{\tau_n}\bigg)^{\alpha}\Big\{|\langle R(u_{h,\tau})\,;\,w^{(n)}\rangle| + |\langle R_\tau(u_{h,\tau})\,;\,w^{(n)}\rangle| \\
&\qquad\qquad + |\langle R_D(u_{h,\tau})\,;\,w^{(n)}\rangle|\Big\}dt \\
&\quad\le \sqrt{c_\kappa}c_{h,\ell}\sqrt{\alpha+1}\tau_n^{1/2}\eta_h^{(n)}\|u-u_{h,\tau}\|_{X(t_{n-1},t_n)} + c_8\bigg|\theta-\frac{\alpha+1}{\alpha+2}\bigg|\tau_n\Big(\eta_h^{(n)}\Big)^2 \\
&\qquad + c_9\bigg|\theta-\frac{\alpha+1}{\alpha+2}\bigg|\tau_n^{1/2}\eta_h^{(n)}\bigg\{\|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)} + \|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad\qquad + \tau_n\Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big] \\
&\qquad\qquad + \int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\bigg\}^{1/2} \\
&\qquad + c_{h,\ell}\sqrt{\alpha+1}\tau_n^{1/2}\eta_h^{(n)} \\
&\qquad\quad \times\bigg\{\|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad\qquad + \tau_n\Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big] + \int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\bigg\}^{1/2}.
\end{aligned}$$

Young's inequality now proves (3.69), with a constant c_{10} that depends on c_κ, $c_{h,\ell}$ and c_9. □

We now intend to subtract the first term on the right hand side of (3.69) and the divide by $\tau_n^{1/2}\eta_h^{(n)}$. In order to estimate the included constants, we need another auxiliary result:

4. Claim. *With the choice*

$$\alpha := \frac{2c_8(2\theta-1)}{2c_8(1-\theta)+1}$$

there holds $\alpha\ge 0$,

$$\frac{\alpha+1}{\alpha+2}\le\theta,\quad c_8\bigg|\theta-\frac{\alpha+1}{\alpha+2}\bigg|\le\frac12,\ \textit{and}\ \sqrt{2\alpha+1}\le\sqrt{2}\sqrt{c_8+1}. \tag{3.70}$$

Recalling that $1/2\le\theta\le 1$, it follows that $\alpha\ge 0$ since both brackets in the definition are non-negative. Moreover, if we insert the definition of α, $1/\theta\le 2$ implies

$$\frac{\alpha+1}{\alpha+2} = \frac{2c_8(2\theta-1)+2c_8(1-\theta)+1}{2c_8(2\theta-1)+4c_8(1-\theta)+2} = \frac{2c_8\theta+1}{2c_8+2} = \theta\frac{2c_8+1/\theta}{2c_8+2}\le\theta$$

prove the first inequality of (3.70). Furthermore, there hold

$$c_8\bigg|\theta-\frac{\alpha+1}{\alpha+2}\bigg| = c_8\bigg|\frac{2c_8\theta+2\theta-2c_8\theta-1}{2c_8+2}\bigg| \le c_8\bigg|\frac{2\theta-1}{2c_8}\bigg|\le\frac12$$

as well as

$$\sqrt{\alpha+1}\le\sqrt{\alpha+2} = \sqrt{\frac{2c_8(2\theta-1)+4c_8(1-\theta)+2}{2c_8(1-\theta)+1}} = \sqrt{\frac{2c_8+2}{2c_8(1-\theta)+1}}\le\sqrt{2}\sqrt{c_8+1},$$

which proves the claim. □

Starting from (3.69) we now use (3.70)

$$\begin{aligned}\tau_n\Big(\eta_h^{(n)}\Big)^2 \leq & c_8\left|\theta-\frac{\alpha+1}{\alpha+2}\right|\tau_n\Big(\eta_h^{(n)}\Big)^2 + c_{10}\left[\sqrt{\alpha+1}+\left|\theta-\frac{\alpha+1}{\alpha+2}\right|\right]\tau_n^{1/2}\eta_h^{(n)}\times\\ &\times\Bigg\{\|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)}+\|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})}\\ &\quad+\tau_n\Big[\Big(\eta_N^{(n)}\Big)^2+(1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big]+\int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\Bigg\}^{1/2}\\ \leq & \frac{1}{2}\tau_n\Big(\eta_h^{(n)}\Big)^2+c_{10}\tau_n^{1/2}\eta_h^{(n)}\left[\sqrt{2}\sqrt{c_8+1}+\frac{1}{2}\right]\\ &\times\Bigg\{\|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)}+\|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})}\\ &\quad+\tau_n\Big[\Big(\eta_N^{(n)}\Big)^2+(1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big]+\int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\Bigg\}^{1/2}.\end{aligned}$$

Subtracting the first term on the right hand side and dividing by $\tau_n^{1/2}\eta_h^{(n)}$, we obtain

$$\begin{aligned}\frac{1}{2}\tau_n^{1/2}\eta_h^{(n)} \lesssim & \Bigg\{\|u-u_{h,\tau}\|^2_{X(t_{n-1},t_n)}+\|f-f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})}\\ &\quad+\tau_n\Big[\Big(\eta_N^{(n)}\Big)^2+(1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot g(s)\|^2_{\Gamma_N}ds\Big]+\int_{t_{n-1}}^{t_n}\big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2dt\Bigg\}^{1/2}.\end{aligned}$$

The previous estimate together with (3.67) now prove (3.66). ■

3.6.1 Estimating $|||\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|||_*$

The term $|||\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|||_*$ from (3.63) and (3.66) is not feasible for a computable error estimator. For the lower bound (3.66) we may simply drop this non-negative term. To derive a computable upper bound for $|||\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|||_*$ we have to distinguish two cases. Due to $\vec{c}\in\mathcal{C}(0,T;L^\infty(\Omega)^2)$, c.f. Assumption $(P1)$, there is a constant $c_{\vec{c}}$, such that

$$\max_{1\leq n\leq N}\|\vec{c}^{(n),\theta}\|_{L^\infty(\Omega)}\leq c_{\vec{c}}\lambda \tag{3.71}$$

where λ is the constant introduced in $(P2)$.

Small Convection

If $c_{\vec{c}}$ is small with respect to λ, the term $|||\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|||_*$ yields a relatively small contribution in (3.63). Then, we obtain a good estimate by the following lemma:

Lemma 3.17. *There is a constant that only depends on $c_{\vec{c}}$ from* (3.71)*, the Poincaré constant c_P and on the diameter of Ω such that*

$$|||\vec{c}^{(n),\theta}\cdot\nabla(u_h^{(n)}-u_h^{(n-1)})|||_*\lesssim|||u_h^{(n)}-u_h^{(n-1)}||| \tag{3.72}$$

for all integers n with $1\leq n\leq N$.

Proof. For $v, w \in H^1_D$, Cauchy's inequality and the Friedrichs inequality imply

$$(\vec{c}^{(n),\theta} \cdot \nabla v \,;\, w) \leq \|\vec{c}^{(n),\theta} \cdot \nabla v\| \|w\| \overset{(3.71)}{\leq} c_{\vec{c}} \lambda \lambda^{-1/2} \{\lambda \|\nabla v\|^2\}^{1/2} c_P \operatorname{diam}(\Omega) \lambda^{-1/2} \{\lambda \|\nabla w\|^2\}^{1/2} \lesssim |\!|\!| v |\!|\!| |\!|\!| w |\!|\!|.$$

We recall the definition of the dual norm, c.f. Definition 10, divide by $|\!|\!| w |\!|\!|$ and take the supremum. Inserting $v = u_h^{(n)} - u_h^{(n-1)}$ in the previous inequality then proves (3.72). ■

Corollary 3.18 (Upper and Lower Bound for Small Convection). *Assume that $c_{\vec{c}} \ll \lambda$. Then, the error between the solution u of the weak form (2.26) and the discrete solution $u_{h,\tau}$ of (2.29) and (2.31) is bounded from above by*

$$\begin{aligned} &\|u - u_{h,\tau}\|^2_{X(0,T)} \\ &\quad \lesssim \|u_0 - \Pi_0 u_0\|^2 + \sum_{n=1}^N \tau_n \Big[\Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2 + |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 \Big] + \|f - f_{h,\tau}\|^2_{L^2(0,T;H^{-1})} \\ &\qquad + \sum_{n=1}^N \tau_n \Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|^2_{\Gamma_N} ds \Big] + \sum_{n=1}^N \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt \end{aligned} \tag{3.73}$$

with a constant depending only on the shape regularity constant c_S, the transition constant c_T, the constants κ from $(P2)$ and c_r from $(P4)$, on the Poincaré constant c_P, on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$, and on the diameter of Ω. For spatially constant data D, $\vec{c}$ and r the error is bounded from below by

$$\begin{aligned} &\tau_n \Big\{ \Big(\eta_h^{(n)}\Big)^2 + |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 \Big\} \\ &\quad \lesssim \|u - u_{h,\tau}\|^2_{X(t_{n-1},t_n)} + \|f - f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\ &\qquad + \tau_n \Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|^2_{\Gamma_N} ds \Big] + \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt \end{aligned} \tag{3.74}$$

with a constant depending on the shape regularity constant c_S, the transition constant c_T, and the constants κ from $(P2)$ and c_r from $(P4)$.

Proof. The first equation (3.73) follows from (3.63) and (3.72). The second equation is obtained from (3.66) by dropping the term $|\!|\!| \vec{c}^{(n),\theta} \cdot \nabla (u_h^{(n)} - u_h^{(n-1)}) |\!|\!|_*$. ■

Large Convection

If $\max\limits_{1 \leq n \leq N} \|\vec{c}^{(n),\theta}\|_{L^\infty(\Omega)} \gg \lambda$, the resulting constant $c_{\vec{c}}$ in (3.71) is very large. In this case the convection is the dominant process and thus, the data function $\vec{c}$ somehow has to be included in the estimates. We approach this problem by solving a suited elliptic problem in the space $\mathcal{S}^1_D(\widetilde{\mathcal{T}}^{(n)})$, denoting the P^1-Finite-Element space for the triangulation $\widetilde{\mathcal{T}}^{(n)}$ which, again, is a common refinement of $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$. We assume in the following, that the discrete solutions $u_h^{(n)}$ and $u_h^{(n-1)}$ are given. Analogously to the error estimator for $\mathcal{T}^{(n)}$, c.f. Definition 23, we now introduce an error estimator for $\widetilde{\mathcal{T}}^{(n)}$:

Definition 29. Let $\widetilde{u}_h^{(n)} \in \mathcal{S}_D^1(\widetilde{\mathcal{T}}^{(n)})$ be the unique solution of the discrete elliptic problem

$$\lambda(\nabla v_h \,;\, \nabla \widetilde{u}_h^{(n)}) + \beta(v_h \,;\, \widetilde{u}_h^{(n)}) = (v_h \,;\, \bar{c}_h^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})) \quad \forall v_h \in \mathcal{S}_D^1(\widetilde{\mathcal{T}}^{(n)}), \tag{3.75}$$

where λ, β are defined in the Assumptions $(P2)$ and $(P3)$, respectively. Then, we define the **convection error estimator** $\eta_{\vec{c}}^{(n)}$ by

$$\begin{aligned}\eta_{\vec{c}}^{(n)} :=& \Bigg\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \left[\| \bar{c}_h^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)}) - \beta \widetilde{u}_h^{(n)} \|_{\widetilde{K}}^2 + \| (\bar{c}^{(n),\theta} - \bar{c}_h^{(n),\theta}) \cdot \nabla(u_h^{(n)} - u_h^{(n-1)}) \|_{\widetilde{K}}^2 \right] \\ &+ \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)} \setminus \Gamma_D} \lambda^{-1/2} \alpha_{\widetilde{E}} \| [\vec{n}_{\widetilde{E}} \cdot \nabla \widetilde{u}_h^{(n)}]_{\widetilde{E}} \|_{\widetilde{E}}^2 \Bigg\}^{1/2}\end{aligned} \tag{3.76}$$

where $\alpha_{\widetilde{K}}, \alpha_{\widetilde{E}}$ are the local parameters, c.f. Definition 22. The entity $[\cdot]_{\widetilde{E}}$ denotes the edge jump over the edge $\widetilde{E}$ from Equation (3.21).

Remark. The constant coefficient functions

$$D' := \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}, \; \vec{c}' := \vec{0}, \; r' := \beta \tag{3.77}$$

satisfy the Assumptions $(E1)$-$(E5)$. So, according to the considerations in Section 2.2 the weak form (3.75) allows for a unique solution.

Lemma 3.19. *For each time step $n = 1, \ldots, N$ there holds*

$$|||\bar{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_* \lesssim |||\widetilde{u}_h^{(n)}||| + \eta_{\vec{c}}^{(n)}. \tag{3.78}$$

Provided that $\vec{c}$ is constant with respect to space, there holds

$$|||\widetilde{u}_h^{(n)}||| + \eta_{\vec{c}}^{(n)} \lesssim |||\bar{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_*. \tag{3.79}$$

Both inequalities include constants that depend only on the shape regularity constant c_S and the transition constant c_T but not on $c_{\vec{c}}$.

Proof.
1. Claim. *If $\widetilde{u}^{(n)} \in H_D^1$ denotes the unique solution of the stationary reaction-diffusion equation*

$$\lambda(\nabla v \,;\, \nabla \widetilde{u}^{(n)}) + \beta(v \,;\, \widetilde{u}^{(n)}) = (v \,;\, \bar{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})) \quad \forall v \in H_D^1. \tag{3.80}$$

there holds

$$|||\widetilde{u}^{(n)}||| = |||\bar{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_*. \tag{3.81}$$

Inserting $v = \widetilde{u}^{(n)}$ in (3.80) implies

$$\begin{aligned}
|||\widetilde{u}^{(n)}|||^2 = \lambda(\nabla\widetilde{u}^{(n)} ; \nabla\widetilde{u}^{(n)}) + \beta(\widetilde{u}^{(n)} ; \widetilde{u}^{(n)}) =& (\widetilde{u}^{(n)} ; \vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)})) \\
=& \frac{(\widetilde{u}^{(n)} ; \vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)}))}{|||\widetilde{u}^{(n)}|||} |||\widetilde{u}^{(n)}||| \\
\leq& |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_* |||\widetilde{u}^{(n)}|||.
\end{aligned}$$

On the other hand, Cauchy's inequality implies

$$\begin{aligned}
|||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_* &= \sup_{v \in H^1_D} \frac{(v ; \vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n-1)}))}{|||v|||} \\
&= \sup_{v \in H^1_D \setminus \{0\}} \frac{\lambda(\nabla v ; \nabla\widetilde{u}^{(n)}) + \beta(v ; \widetilde{u}^{(n)})}{|||v|||} \\
&\leq \sup_{v \in H^1_D \setminus \{0\}} \frac{\lambda^{1/2}\|\nabla v\|\lambda^{1/2}\|\nabla\widetilde{u}^{(n)}\| + \beta^{1/2}\|v\|\beta^{1/2}\|\widetilde{u}^{(n)}\|}{|||v|||} \\
&\leq \sup_{v \in H^1_D \setminus \{0\}} \frac{\{\lambda\|\nabla v\|^2 + \beta\|v\|^2\}^{1/2}\{\lambda\|\nabla\widetilde{u}^{(n)}\|^2 + \beta\|\widetilde{u}^{(n)}\|^2\}^{1/2}}{|||v|||} \\
&= \sup_{v \in H^1_D \setminus \{0\}} \frac{|||v|||\widetilde{u}^{(n)}|||}{|||v|||} \\
&= |||\widetilde{u}^{(n)}|||.
\end{aligned}$$

This proves (3.81). □

Analogously to the previous considerations, we have

$$|||\widetilde{u}_h^{(n)}||| \leq |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_*.$$

This and Equation (3.81) together with the triangle inequality lead to

$$\begin{aligned}
&\frac{1}{3}\Big\{|||\widetilde{u}_h^{(n)}||| + |||\widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}|||\Big\} \leq \frac{1}{3}\Big\{|||\widetilde{u}_h^{(n)}||| + |||\widetilde{u}^{(n)}||| + |||\widetilde{u}_h^{(n)}|||\Big\} \leq |||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_*, \\
&|||\vec{c}^{(n),\theta} \cdot \nabla(u_h^{(n)} - u_h^{(n)})|||_* = |||\widetilde{u}_h^{(n)} + \widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}||| \leq \Big\{|||\widetilde{u}_h^{(n)}||| + |||\widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}|||\Big\}.
\end{aligned} \tag{3.82}$$

2. Claim. *The error estimator $\eta_{\vec{c}}^{(n)}$ from (3.76) provides an upper for the error $|||\widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}|||$, i.e.*

$$|||\widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}||| \lesssim \eta_{\vec{c}}^{(n)}, \tag{3.83}$$

and for space-constant data $\vec{c}$, it provides a lower bound

$$\eta_{\vec{c}}^{(n)} \lesssim |||\widetilde{u}^{(n)} - \widetilde{u}_h^{(n)}|||. \tag{3.84}$$

Both estimates include constants that depend on the shape regularity constant c_S and the transition constant c_T.

A proof can be found in Verfürth, [15, Prop. 4.1, p. 488]. Note, that the differing factors to the referred paper arise from the different definition of the energy norm. Moreover, we define the error estimator as sum over all elements and edges, and have to make sure that we count

an edge only once. □

Inequality (3.82) together with (3.83) and (3.84) now prove the lemma. ■

Corollary 3.20 (Upper and Lower Bound for Large Convection). *Assume that $c_{\vec{c}} \gg \lambda$. Then, the error between the solution u of the weak form (2.26) and the discrete solution $u_{h,\tau}$ of (2.29) and (2.31) is bounded from above by*

$$\begin{aligned}
&\|u - u_{h,\tau}\|^2_{X(0,T)} \\
&\lesssim \|u_0 - \Pi_0 u_0\|^2 + \sum_{n=1}^{N} \tau_n \Big[\Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2 + |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + \Big(\eta_{\vec{c}}^{(n)}\Big)^2 + |\!|\!| \widetilde{u}_h^{(n)} |\!|\!|^2 \Big] \\
&\quad + \sum_{n=1}^{N} \tau_n \Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|^2_{\Gamma_N} ds \Big] + \sum_{n=1}^{N} \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt.
\end{aligned} \tag{3.85}$$

Let D,$\vec{c}$ and r be spatially independent. Then, the error is bounded from below by

$$\begin{aligned}
&\tau_n \Big\{ \Big(\eta_h^{(n)}\Big)^2 + |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + \Big(\eta_{\vec{c}}^{(n)}\Big)^2 + |\!|\!| \widetilde{u}_h^{(n)} |\!|\!|^2 \Big\} \\
&\quad \lesssim \|u - u_{h,\tau}\|^2_{X(t_{n-1},t_n)} + \|f - f_{h,\tau}\|^2_{L^2(t_{n-1},t_n;H^{-1})} \\
&\qquad + \tau_n \Big[\Big(\eta_N^{(n)}\Big)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{g}(s)\|^2_{\Gamma_N} ds \Big] + \int_{t_{n-1}}^{t_n} \big[\varrho_\tau^{(n)}(u_{h,\tau})\big]^2 dt
\end{aligned} \tag{3.86}$$

on each interval $(t_{n-1}, t_n]$, $1 \le n \le N$. For both estimates the constants depend only on the shape regularity constant c_S, the transition constant c_T, and the constants κ from $(P2)$ and c_r from $(P4)$. For the upper bound the constant additionally depends on the Poincaré constant c_P, on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$, and on the diameter of Ω.

Proof. The two estimates directly follow from (3.63) and (3.66) with (3.82). ■

To increase the quality of the a posteriori estimates, the last corollary shows that we can solve an additional problem (3.80) on the grid $\widetilde{\mathcal{T}}^{(n)}$ for each time step $1 \le n \le N$ in case of large convection.

3.6.2 Estimating the Dual Norms

In the following we assume the convection to be small. However, the considerations of this section may easily be generalized for problems with large convection. In Corollary 3.18 the terms representing dual norms of functionals in H^{-1} show no way of direct computation. We now estimate these norms by use of weighted L^2-norms. Again, we choose a fixed integer n between 1 and N for this section. Recall the θ-weighted space-time approximation

$$f_{h,\tau} = \theta f_h^{(n)} + (1-\theta) f_h^{(n-1)}.$$

Here, $f_h^{(n)} := \widetilde{\Pi}_h^{(n)} f^{(n)}$ where $\widetilde{\Pi}_h^{(n)}$denotes the L^2-projection onto $\mathcal{S}^1_D(\widetilde{\mathcal{T}}^{(n)})$. Then, we define the **$f$-data error estimator** by

$$\eta_f^{(n)} := \Big\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}} \Big[\theta^2 \|f(t_n) - f_h^{(n)}\|^2_{\widetilde{K}} + (1-\theta)^2 \|f(t_{n-1}) - f_h^{(n-1)}\|^2_{\widetilde{K}} \Big] \Big\}^{1/2}. \tag{3.87}$$

Lemma 3.21. *Let $f \in L^2(0,T;L^2(\Omega))$, $\dot{f} \in L^2(0,T;L^2(\Omega))$, $\widetilde{\mathcal{T}}^{(n)}$ a common refinement of $\mathcal{T}^{(n)}$ and $\mathcal{T}^{(n-1)}$ with element parameters $\alpha_{\widetilde{K}}$, θ the time discretization parameter and τ_n the length of the time interval $(t_{n-1},t_n]$ with $n=1,\dots,N$. Then, we have the following estimate*

$$\|f-f_{h,\tau}\|^2_{L^2(0,T;H^{-1})} \lesssim \sum_{n=1}^{N} \tau_n \left[\left(\eta_f^{(n)} \right)^2 + (1+\theta)^2 \tau_n \int_{t_{n-1}}^{t_n} \|\dot{f}(s)\|^2 ds \right] \tag{3.88}$$

with a constant that depends only on the shape regularity constant c_S, the transition constant c_T, on the Pioncaré constant c_P, on λ from $(P2)$ and β from $(P3)$, and on the diameter of Ω.

Proof.

1. Claim. *We have the following representation*

$$f(t) = \theta f(t_n) + (1-\theta) f(t_{n-1}) + \int_{t_{n-1}}^{t} \dot{f}(s) ds - \theta \int_{t_{n-1}}^{t_n} \dot{f}(s) ds. \tag{3.89}$$

The right hand side of (3.89) yields

$$\theta f(t_n) + (1-\theta) f(t_{n-1}) + f(t) - f(t_{n-1}) - \theta f(t_n) + \theta f(t_{n-1}) = f(t).$$

This proves the claim. □

2. Claim. *There holds*

$$\left\{ \sup_{v \in H^1_D \setminus \{0\}} \frac{(f(t_n) - f_h^{(n)} \, ; \, v)}{\|\!|v|\!\|} \right\}^2 \lesssim \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}} \|f(t_n) - f_h^{(n)}\|^2_{\widetilde{K}} \tag{3.90}$$

with a constant depending on the shape regularity constant c_S, the transition constant c_T, on the Pioncaré constant c_P and on the diameter of Ω.

Since $f_h^{(n)} := \widetilde{\Pi}_h^{(n)} f^{(n)}$, the function $f(t_n) - f_h^{(n)}$ belongs to the orthogonal space of $\mathcal{S}^1_D(\widetilde{\mathcal{T}}^{(n)})$ with respect to the L^2-scalar product. Thus, $(f(t_n) - f_h^{(n)} \, ; \, \widetilde{\Pi}_h^{(n)} v) = 0$ for $v \in H^1_D$. Consequently,

$$(f(t_n) - f_h^{(n)} \, ; \, v) = (f(t_n) - f_h^{(n)} \, ; \, v - \widetilde{\Pi}_h^{(n)} v) \leq \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \|f(t_n) - f_h^{(n)}\|_{\widetilde{K}} \|v - \widetilde{\Pi}_h^{(n)} v\|_{\widetilde{K}}.$$

Using the best approximation quality of $\widetilde{\Pi}_h^{(n)}$ gives $\|v - \widetilde{\Pi}_h^{(n)} v\|_{\widetilde{K}} \leq \|v - I_h^{(n)} v\|_{\widetilde{K}}$ where $I_h^{(n)}$ denotes the Clément operator of Definition 24. With (3.25) and Cauchy's inequality, we obtain

$$\begin{aligned}
(f(t_n) - f_h^{(n)} \, ; \, v) &\leq \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \|f(t_n) - f_h^{(n)}\|_{\widetilde{K}} \|v - I_h^{(n)} v\|_{\widetilde{K}} \\
&\lesssim \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}} \|f(t_n) - f_h^{(n)}\|_{\widetilde{K}} \|\!|v|\!\|_{\omega_{\widetilde{K}}} \\
&\leq \left\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha^2_{\widetilde{K}} \|f(t_n) - f_h^{(n)}\|^2_{\widetilde{K}} \right\}^{1/2} \cdot \left\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \|\!|v|\!\|^2_{\omega_{\widetilde{K}}} \right\}^{1/2}.
\end{aligned}$$

Analogously to the proof of Lemma 3.10, there is a constant depending on the shape regularity constant c_S and the transition constant c_T such that

$$\left\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \|\!|v|\!\|^2_{\omega_{\widetilde{K}}} \right\}^{1/2} \lesssim \left\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \|\!|v|\!\|^2_{\widetilde{K}} \right\}^{1/2} = \|\!|v|\!\|$$

which proves (3.90). □

The representation (3.89) now further implies

$$\begin{aligned}\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(f(t)-f_{h,\tau}\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2 \leq&\theta^2\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(f(t_n)-f_h^{(n)}\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2\\ &+(1-\theta)^2\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(f(t_{n-1})-f_h^{(n-1)}\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2\\ &+\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(\int_{t_{n-1}}^t\dot f(s)ds-\theta\int_{t_{n-1}}^{t_n}\dot f(s)ds\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2.\end{aligned}$$

With (3.90) this yields

$$\begin{aligned}\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(f(t)-f_{h,\tau}\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2 \lesssim&\theta^2\sum_{\widetilde K\in\widetilde{\mathcal T}^{(n)}}\alpha_{\widetilde K}^2\|f(t_n)-f_h^{(n)}\|_{\widetilde K}^2\\ &+(1-\theta)^2\sum_{\widetilde K\in\widetilde{\mathcal T}^{(n)}}\alpha_{\widetilde K}^2\|f(t_{n-1})-f_h^{(n-1)}\|_{\widetilde K}^2\\ &+\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{\|\int_{t_{n-1}}^t\dot f(s)ds-\theta\int_{t_{n-1}}^{t_n}\dot f(s)ds\|\,\|v\|}{|\!|\!|v|\!|\!|}\right\}^2\\ \leq&\left(\eta_f^{(n)}\right)^2+(1+\theta)^2\left\{\int_{t_{n-1}}^{t_n}\|\dot f(s)\|ds\right\}^2\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{\|v\|}{|\!|\!|v|\!|\!|}\right\}^2.\end{aligned}$$

With the Friedrichs inequality (3.1) we have the following estimate

$$\|v\|\leq\min\{c_P\,\mathrm{diam}(\Omega)\lambda^{-1/2},\beta^{-1/2}\}|\!|\!|v|\!|\!| \tag{3.91}$$

for $v\in H^1_D$. Finally, with Cauchy's inequality,

$$\left\{\sup_{v\in H^1_D\setminus\{0\}}\frac{(f(t)-f_{h,\tau}\,;\,v)}{|\!|\!|v|\!|\!|}\right\}^2\lesssim\left(\eta_f^{(n)}\right)^2+(1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot f(s)\|^2ds$$

where none of the terms on the right hand side depends on t. Piecewise integration over the time intervals $[t_{n-1},t_n]$ now proves (3.88). ■

In Equation (3.88) the f-data error estimator on the right hand side correlates to the Verfürth-type error estimator – both are of first order of the diameter of the elements of $\widetilde{\mathcal T}^{(n)}$.
We now focus on the other data functions. Recall the definition of the temporal data error estimator

$$\varrho_\tau^{(n)}(u_{h,\tau})=\sup_{v\in H^1_D\setminus\{0\}}\frac{((D^{(n),\theta}-D)\nabla u_{h,\tau}\,;\,\nabla v)}{|\!|\!|v|\!|\!|}+|\!|\!|(\vec c^{(n),\theta}-\vec c)\cdot\nabla u_{h,\tau}+(r^{(n),\theta}-r)u_{h,\tau}|\!|\!|_*,$$

as well as the θ-weighted time approximations

$$\begin{aligned}D^{(n),\theta}&=\theta D^{(n)}+(1-\theta)D^{(n-1)},\\ c^{(n),\theta}&=\theta\vec c^{(n)}+(1-\theta)\vec c^{(n-1)},\\ r^{(n),\theta}&=\theta r^{(n)}+(1-\theta)r^{(n-1)}.\end{aligned}$$

Corollary 3.22. *Let $\dot{D} \in L^2(0,T;L^2(\Omega)^{2\times 2})$, $\dot{\vec{c}} \in L^2(0,T;L^2(\Omega)^2)$ and $\dot{r} \in L^2(0,T;L^2(\Omega))$, θ the time discretization parameter and τ_n the local time step length. Then, we have the following estimate*

$$\begin{aligned}\int_{t_{n-1}}^{t_n} \left[\varrho_\tau^{(n)}(u_{h,\tau})\right]^2 dt \lesssim (1+\theta)^2\tau_n \Big\{ &\lambda^{-1}\int_{t_{n-1}}^{t_n} \|\dot{D}(t)\nabla u_{h,\tau}\|^2 dt \\ &+ \int_{t_{n-1}}^{t_n} \|\dot{\vec{c}}(t)\cdot\nabla u_{h,\tau}\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot{r}(t)u_{h,\tau}(\overline{t}_n)\|^2 dt\Big\}\end{aligned} \tag{3.92}$$

where the constant depends only on the Pioncaré constant c_P, on λ from $(P2)$ and β from $(P3)$, and on the diameter of Ω.

Proof. From (3.89), we get the following representation

$$D^{(n),\theta} - D = -\int_{t_{n-1}}^{t} \dot{D}(s)ds + \theta\int_{t_{n-1}}^{t_n}\dot{D}(s)ds$$

and analogously for the terms including $\vec{c}$ and r for a fixed time interval $[t_{n-1},t_n]$. Similar considerations as in the proof of Lemma 3.21 show

$$\begin{aligned}&\int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{((D^{(n),\theta}-D(t))\nabla u_{h,\tau}\,;\nabla v)}{|\!|\!|v|\!|\!|}\right\}^2 dt \\ &\quad\le \int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{\|(D^{(n),\theta}-D(t))\nabla u_{h,\tau}\|\,\|\nabla v\|}{|\!|\!|v|\!|\!|}\right\}^2 dt \\ &\quad\le \lambda^{-1}\int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{\|(-\int_{t_{n-1}}^{t}\dot{D}(s)ds+\theta\int_{t_{n-1}}^{t_n}\dot{D}(s)ds)\nabla u_{h,\tau}\|\,|\!|\!|v|\!|\!|}{|\!|\!|v|\!|\!|}\right\}^2 dt.\end{aligned}$$

The gradient $\nabla u_{h,\tau}$ is constant with respect to time on $(t_{n-1},t_n]$. Thus, we obtain

$$\int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{((D^{(n),\theta}-D(t))\nabla u_{h,\tau}\,;\nabla v)}{|\!|\!|v|\!|\!|}\right\}^2 dt \lesssim \lambda^{-1}(1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot{D}(t)\nabla u_{h,\tau}\|^2 dt.$$

We now employ analogous considerations to obtain estimates for the other terms on the left hand side of (3.92). First, estimating $\|v\|$ by (3.91) leads to

$$\int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{((c^{(n),\theta}-\vec{c}(t))\cdot\nabla u_{h,\tau}\,;v)}{|\!|\!|v|\!|\!|}\right\}^2 dt \lesssim (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot{\vec{c}}(t)\cdot\nabla u_{h,\tau}\|^2 dt.$$

The reaction term $|\!|\!|(r^{(n),\theta}-r)u_{h,\tau}|\!|\!|_*$ includes the discrete solution $u_{h,\tau}$, which is affine with respect to time. With $\overline{t}_n := (t_n+t_{n-1})/2$ we obtain

$$\begin{aligned}&\int_{t_{n-1}}^{t_n}\left\{\sup_{v\in H_D^1\setminus\{0\}} \frac{((r^{(n),\theta}-r(t))u_{h,\tau}\,;v)}{|\!|\!|v|\!|\!|}\right\}^2 dt \\ &\quad\lesssim \int_{t_{n-1}}^{t_n}\left\|\Big\{-\int_{t_{n-1}}^{t}\dot{r}(s)ds+\theta\int_{t_{n-1}}^{t_n}\dot{r}(s)ds\Big\}u_{h,\tau}(t)\right\|^2 dt \\ &\quad\lesssim (1+\theta)^2\int_{t_{n-1}}^{t_n}\int_{t_{n-1}}^{t_n}\|\dot{r}(s)u_{h,\tau}(t)\|^2 ds dt \\ &\quad\lesssim (1+\theta)^2\tau_n\int_{t_{n-1}}^{t_n}\|\dot{r}(t)u_{h,\tau}(\overline{t}_n)\|^2 dt\end{aligned}$$

which proves the corollary. ■

3.6.3 Final Upper Bound

We assume small convection. All terms in (3.73) may now be estimated by computable entities. So, we finally state the upper bound for the error. For abbreviation, we define the **temporal error estimator** by

$$\eta_\tau^{(n)} := \Bigg\{ \|\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + (1+\theta)^2 \tau_n \Bigg[\int_{t_{n-1}}^{t_n} \|\dot f(t)\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot g(t)\|_{\Gamma_N}^2 dt + \lambda^{-1} \int_{t_{n-1}}^{t_n} \|\dot D(t) \nabla u_{h,\tau}\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot{\vec c}(t) \cdot \nabla u_{h,\tau}\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot r(t) u_{h,\tau}(\overline{t}_n)\|^2 dt \Bigg] \Bigg\}^{1/2}. \tag{3.93}$$

Note that for data, which is not time dependent, all terms but the first on the right hand side of this definition are trivial.

Corollary 3.23 (Final Upper Bound). *Assume that $c_{\vec c} \ll \lambda$. Then, the error between the solution u of the weak form (2.26) and the discrete solution $u_{h,\tau}$ of (2.29) and (2.31) is bounded from above by*

$$\begin{aligned} &\|u - u_{h,\tau}\|_{X(0,T)}^2 \\ &\lesssim \|u_0 - \Pi_0 u_0\|^2 + \sum_{n=1}^N \tau_n \left[\left(\eta_\tau^{(n)}\right)^2 + \left(\eta_h^{(n)}\right)^2 + \left(\Theta_h^{(n)}\right)^2 + \left(\eta_f^{(n)}\right)^2 + \left(\eta_N^{(n)}\right)^2 \right] \end{aligned} \tag{3.94}$$

with a constant that depends only on the shape regularity constant c_S, the transition constant c_T, the constants κ from $(P2)$ and c_r from $(P4)$, on the Poincaré constant c_P, on the shape of the patches ω_a of all nodes $a \in \mathcal{N}^{(n)}$, on λ from $(P2)$ and β from $(P3)$, and on the diameter of Ω.

Proof. This estimate follows from (3.73) with (3.88) and (3.92). ■

3.7 Space-Time Adaptive Algorithm

The considerations of the previous section lead to an upper bound (3.94) which consists of various decomposed contributions. To obtain a global error estimator, we first recall the definitions of these terms: The Verfürth-type error estimator $\eta_h^{(n)}$ and the data error estimator $\Theta_h^{(n)}$ read

$$\eta_h^{(n)} = \Bigg\{ \sum_{\widetilde K \in \widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde K}^2 \|R_{\widetilde K}\|_{\widetilde K}^2 + \sum_{\widetilde E \in \widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2} \alpha_{\widetilde E} \|R_{\widetilde E}\|_{\widetilde E}^2 \Bigg\}^{1/2},$$

$$\Theta_h^{(n)} = \Bigg\{ \sum_{\widetilde K \in \widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde K}^2 \|D_{\widetilde K}\|_{\widetilde K}^2 + \sum_{\widetilde E \in \widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2} \alpha_{\widetilde E} \|D_{\widetilde E}\|_{\widetilde E}^2 \Bigg\}^{1/2},$$

respectively, where

$$
\begin{aligned}
R_{\widetilde{K}} =& \Big[f_{h,\tau} - \frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n} + \operatorname{div}\{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} \\
& - \bar{c}_h^{(n),\theta} \cdot \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) - r_h^{(n),\theta}(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) \Big] \Big|_{\widetilde{K}} \\
R_{\widetilde{E}} =& \begin{cases} -[\vec{n} \cdot \{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ g_{h,\tau} - \vec{n} \cdot \{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D. \end{cases} \\
D_{\widetilde{K}} =& \{ - \operatorname{div}\{(D_h^{(n),\theta} - D^{(n),\theta}) \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} \\
& + (\bar{c}_h^{(n),\theta} - \bar{c}^{(n)}) \cdot \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) + (r_h^{(n),\theta} - r^{(n),\theta})(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) \}|_{\widetilde{K}} \\
D_{\widetilde{E}} =& \begin{cases} [\vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta}) \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ \vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta}) \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D, \end{cases}
\end{aligned}
$$

Moreover, the Neumann boundary error estimator $\eta_N^{(n)}$ and the f-data error estimator $\eta_f^{(n)}$ yield

$$
\begin{aligned}
\eta_N^{(n)} &= \lambda^{-1/4} \Big\{ \sum_{\widetilde{E} \in \widetilde{\mathcal{E}}_{\Gamma_N}} \alpha_{\widetilde{E}} \Big[\theta^2 \|g(t_n) - g_h^{(n)}\|_{\widetilde{E}}^2 + (1-\theta)^2 \|g(t_{n-1}) - g_h^{(n-1)}\|_{\widetilde{E}}^2 \Big] \Big\}^{1/2}, \\
\eta_f^{(n)} &= \Big\{ \sum_{\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \Big[\theta^2 \|f(t_n) - f_h^{(n)}\|_{\widetilde{K}}^2 + (1-\theta)^2 \|f(t_{n-1}) - f_h^{(n-1)}\|_{\widetilde{K}}^2 \Big] \Big\}^{1/2},
\end{aligned}
$$

respectively. The preceding terms all include local parameters $\alpha_S = \min\{h_S \lambda^{-1/2}, \beta^{-1/2}\}$, with $S \in \{\widetilde{K}, \widetilde{E}\}$, which correspond to contributions of order $\mathcal{O}(\alpha_S) \sim \mathcal{O}(h_S)$. Thus, we may define the **spatial error estimator** by

$$
\eta_S^{(n)} := \Big\{ \Big(\eta_h^{(n)}\Big)^2 + \Big(\Theta_h^{(n)}\Big)^2 + \Big(\eta_f^{(n)}\Big)^2 + \Big(\eta_N^{(n)}\Big)^2 \Big\}^{1/2}. \tag{3.95}
$$

Finally, we recall the temporal error estimator as

$$
\begin{aligned}
\eta_\tau^{(n)} =& \Big\{ |\!|\!| u_h^{(n)} - u_h^{(n-1)} |\!|\!|^2 + (1+\theta)^2 \tau_n \Big[\int_{t_{n-1}}^{t_n} \|\dot{f}(t)\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot{g}(t)\|_{\Gamma_N}^2 dt \\
& + \lambda^{-1} \int_{t_{n-1}}^{t_n} \|\dot{D}(t) \nabla u_{h,\tau}\|^2 dt + \int_{t_{n-1}}^{t_n} \|\dot{\bar{c}}(t) \cdot \nabla u_{h,\tau}\|^2 dt \\
& + \int_{t_{n-1}}^{t_n} \|\dot{r}(t) u_{h,\tau}(\overline{t}_n)\|^2 dt \Big] \Big\}^{1/2}.
\end{aligned}
$$

After these preparations, we define the **time-local error estimators** by

$$
\eta^{(n)} := \begin{cases} \|u_0 - \Pi_{h,0} u_0\| & \text{for } n = 0, \\ \Big\{ \Big(\eta_S^{(n)}\Big)^2 + \Big(\eta_\tau^{(n)}\Big)^2 \Big\}^{1/2} & \text{for } n = 1, \dots, N, \end{cases}
$$

and the **error estimator** by

$$\eta := \Big\{ \sum_{n=0}^{N} \tau_n \Big(\eta^{(n)}\Big)^2 \Big\}^{1/2},$$

where τ_n is the local time step size and N denotes the number of times steps.
We heuristically assume the spatial error estimator to be only weakly dependent on the time step size τ_n. On the other hand we presume, that the temporal error estimator does not change crucially if we refine or coarsen a triangulation. These assumptions lead to an adaptive strategy, where both spatial an temporal discretizations are controlled separately. For a given local tolerance ε we thus obtain the time step size τ_n from an arbitrary first guess time step size $\hat{\tau}_n$ by

$$\tau_n = \hat{\tau}_n \frac{\varepsilon}{\hat{\eta}_\tau^{(n)}}.$$

Here, $\hat{\eta}_\tau^{(n)}$ denotes the temporal error estimator computed with use of $\hat{\tau}_n$.
For a given time step n, the Verfürth-type error estimator, the data error estimator $\Theta_h^{(n)}$, the f-data error estimator $\eta_f^{(n)}$ and the Neumann boundary error estimator $\eta_N^{(n)}$, all include sums over all elements respectively edges of the triangulation $\widetilde{\mathcal{T}}^{(n)}$. Thus, we may take the value of the element contribution to identify the triangles of $\widetilde{\mathcal{T}}^{(n)}$ whose refinement would yield the highest decrease in the spatial error estimator (and hopefully of the error) for this time step. For an element $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$, we consider the indicator to include three edge contributions of its edges. Hence, for $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ with edges $\widetilde{E}_1, \widetilde{E}_2, \widetilde{E}_3 \in \widetilde{\mathcal{E}}^{(n)}$, we define the **refinement indicator** by

$$\begin{aligned}\eta_{\widetilde{K}} :=& \Big\{ \alpha_{\widetilde{K}}^2 \big\{ \|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \|D_{\widetilde{K}}\|_{\widetilde{K}}^2 + \theta^2 \|f(t_n) - f_h^{(n)}\|_{\widetilde{K}}^2 + (1-\theta)^2 \|f(t_{n-1}) - f_h^{(n-1)}\|_{\widetilde{K}}^2 \big\} \\ &+ \sum_{i=1}^{3} \lambda^{-1/2} \alpha_{\widetilde{E}_i} \big\{ \|R_{\widetilde{E}_i}\|_{\widetilde{E}_i}^2 + \|D_{\widetilde{E}_i}\|_{\widetilde{E}_i}^2 + \theta^2 \|g(t_n) - g_h^{(n)}\|_{\widetilde{E}_i}^2 \\ &\qquad + (1-\theta)^2 \|g(t_{n-1}) - g_h^{(n-1)}\|_{\widetilde{E}_i}^2 \big\} \Big\}^{1/2},\end{aligned} \tag{3.96}$$

where the functions g, g_h are trivial for all non-Neumann edges. In the following adaptive algorithm, we will use the so-called bulk criterion for refinement and coarsening: An element $K \in \widetilde{\mathcal{T}}^{(n)}$ is marked for red-refinement, if

$$\eta_K > \varrho \max_{\widetilde{K} \in \mathcal{T}^{(n)}} (\eta_{\widetilde{K}}),$$

where $\varrho \in [0,1]$. Analogously, an element is marked for red-coarsening if

$$\eta_K < \varrho \operatorname{median}_{\widetilde{K} \in \mathcal{T}^{(n)}} (\eta_{\widetilde{K}}).$$

We choose $\varrho = 1/2$, which is commonly considered to be a good compromise between convergence rate and efficiency. Note that we employ the matrix equation (2.32) to obtain the discrete solution $u_{h,\tau}(t_n)$ for a given time step t_n. Finally, we state the **space-time adaptive algorithm**[5]:

[5]The algorithm additionally expects two global input parameters: the time discretization parameter $\theta \in [1/2, 1]$, and the bulk criterion parameter $\varrho \in [0, 1]$.

ALGORITHM 1.

Let $\widehat{\mathcal{T}}_0$ be a regular triangulation, $u_0 \in H^1_D(\Omega)$ and $T, \varepsilon \in \mathbb{R}_+$. Then, we consider the following global strategy to obtain the discrete solution $u_{h,\tau}(T)$:

Initialization:

(o.1) Uniform red-refine $\widehat{\mathcal{T}}_0$ as long as $\|u_0 - \Pi_{h,0} u_0\| > \varepsilon$ to obtain $\mathcal{T}^{(0)}$.

(o.2) Set $n = 1$, $t_0 = 0$, $\widehat{\mathcal{T}}^{(1)} := \mathcal{T}^{(0)}$.

(o.3) Set an initial time step length $\widehat{\tau}_1 = \min\{T/2, \varepsilon\}$.

(o.4) Compute $\eta^{(0)} = \|u_0 - \Pi_{h,0} u_0\|$.

***Loop with input:** $\mathcal{T}^{(n-1)}$, $\widehat{\mathcal{T}}^{(n)}$, $u_{h,\tau}(t_{n-1})$, $\widehat{\tau}_n$. Additional input: τ_{n-1} if $n > 1$.*

(i) Define first-guess time step $\widehat{t}_n := \min\{t_{n-1} + \widehat{\tau}_n, T\}$.

(ii) Use $\widehat{\mathcal{T}}^{(n)}$, $u_{h,\tau}(t_{n-1})$ and $\widehat{t}_n$ to compute $\widehat{u}_{h,\tau}(\widehat{t}_n)$.

(iii) Use $u_{h,\tau}(t_{n-1})$ and $\widehat{u}_{h,\tau}(\widehat{t}_n)$ to compute $\widehat{\eta}^{(n)}_\tau$.

(iv) Choose time step $t_n := \min\left\{t_{n-1} + (\widehat{t}_n - t_{n-1})\,\varepsilon/\widehat{\eta}^{(n)}_\tau, T\right\}$, which will be fixed from now on. Consequently, $\tau_n := t_n - t_{n-1}$.

(v) Now, repeat the following loop on the spatial computation:

(v.1) Use $\widehat{\mathcal{T}}^{(n)}$, $u_{h,\tau}(t_{n-1})$, and t_n to compute new $\widehat{u}_{h,\tau}(t_n)$.

(v.2) Compute error indicators η_K for all elements $K \in \widehat{\mathcal{T}}^{(n)}$ and obtain spatial error estimator $\eta^{(n)}_S$.

(v.3) Leave loop and goto (vi) provided $\eta_S \le \varepsilon$.

(v.4) Mark elements for refinement according to the bulk criterion $\eta_K > \varrho \max_{K' \in \mathcal{T}^{(n)}} \eta_{K'}$. Moreover, mark element $K \in \widehat{\mathcal{T}}^{(n)}$ for coarsening if $\eta_K < \varrho\, \mathrm{median}_{K' \in \mathcal{T}^{(n)}} \eta_{K'}$ (this information may be used in (ix).

(v.5) Refine marked elements, obtain a new triangulation $\widehat{\mathcal{T}}^{(n)}$ and goto (v.1)).

(vi) Define $\mathcal{T}^{(n)} := \widehat{\mathcal{T}}^{(n)}$ and $u_{h,\tau}(t_n) := \widehat{u}_{h,\tau}(t_n)$.

(vii) Use $u_{h,\tau}(t_{n-1})$ and $u_{h,\tau}(t_n)$ to compute $\eta^{(n)}_\tau$, and thus obtain the time-local error estimator $\eta^{(n)}$ for the time step t_n.

(viii) In the case of $t_n = T$, define $N = n$, stop computation and return $u_{h,\tau}$, $\eta = \left\{\sum_{n=0}^{N} \tau_n \big(\eta^{(n)}\big)^2\right\}^{1/2}$ and other quantities of interest.

(ix) If $\tau_n > \tau_{n-1}$ with $n > 1$, coarsen the marked elements from (v.4) of $\mathcal{T}^{(n)}$ to obtain $\widehat{\mathcal{T}}^{(n+1)}$. Otherwise define $\widehat{\mathcal{T}}^{(n+1)} := \mathcal{T}^{(n)}$.

(x) Define a first-guess time step size $\widehat{\tau}_{n+1} = \tau_n$ for the next time step.

(xi) Update $n = n + 1$ and goto (i).

This strategy is implemented by the function *solveP*, c.f. Appendix A.7.

3.7.1 The Function *etaSpatial*

This function computes the spatial error estimator $\eta_S^{(n)}$, c.f. Equation (3.95). First, the subfunction *getCommonU* computes the values of $u_{h,\tau}(t_{n-1})$ and $u_{h,\tau}(t_n)$ at the vertices of the common refinement $\widetilde{\mathcal{T}}^{(n)}$ (denoted by *CommonGrid*). Second, the L^2-projections onto $\mathcal{S}_D^1(\widetilde{\mathcal{T}}^{(n)})$ of the data functions are computed, by calling the subfunction *l2Projections*. The latter function also computes the double area of the elements of *CommonGrid*, which will be needed throughout.

```
0120   %Obtain values U,Uold at the vertices of CommonGrid
0121   UCommon = getCommonU(OldGrid,Grid,CommonGrid,Uold,U);
0128   %Obtain L2 projections
0129   [Projection,DoubleArea] =l2Projections(CommonGrid,Told,T,Driver,Options);
0130   CommonGrid.DoubleArea = DoubleArea;
```

Now, we collect all data stored in *CommonGrid* required for the computation: the values of $u_{h,\tau}(t_n)$ at the vertices of *CommonGrid*, the vertices itself, the $\widetilde{\mathcal{T}}^{(n)}$-constant gradients at the vertices, etc.

```
0145   %Collect nodes
0146   Nodes = CommonGrid.N4E(nze,1:3);
0147
0148   %Obtain U at vertices
0149   U1 = UCommon(Nodes(:,1),2);
0150   U2 = UCommon(Nodes(:,2),2);
0151   U3 = UCommon(Nodes(:,3),2);
0152   U1old = UCommon(Nodes(:,1),1);
0153   U2old = UCommon(Nodes(:,2),1);
0154   U3old = UCommon(Nodes(:,3),1);
0155
0156   %Obtain UTheta at vertices
0157   UTheta1 = Theta*U1+(1-Theta)*U1old;
0158   UTheta2 = Theta*U2+(1-Theta)*U2old;
0159   UTheta3 = Theta*U3+(1-Theta)*U3old;
0160
0161   %Collect vertices
0162   A = CommonGrid.C4N(CommonGrid.N4E(nze,1),:);
0163   B = CommonGrid.C4N(CommonGrid.N4E(nze,2),:);
0164   C = CommonGrid.C4N(CommonGrid.N4E(nze,3),:);
0165
0166   %Assembly of gradients
0167   N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)]./(DoubleArea*ones(1,2));
0168   N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)]./(DoubleArea*ones(1,2));
0169   N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)]./(DoubleArea*ones(1,2));
0170
0171   NablaUTheta = N1.*(U1*ones(1,2)) +...
0172                 N2.*(U2*ones(1,2)) + ...
0173                 N3.*(U3*ones(1,2));
```

Analogously to Section 2.3.5, we use the quadrature rule specified in the function *options* (see Appendix A.8) and consequently in the variables QNodes, QWeights and QBasisfct which store the coordinates of the quadrature weights, the corresponding quadrature nodes on the reference element $\widehat{K}$, and the values of the three basis functions thereon, respectively. Note that the latter quantity is also the quadrature node in barycentric coordinates.
We first assemble the element residuals

$$R_{\widetilde{K}} = \Big[f_{h,\tau} - \frac{u_h^{(n)} - u_h^{(n-1)}}{\tau_n} + \operatorname{div}\{D_h^{(n),\theta} \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)})\} \\ - \vec{c}_h^{(n),\theta} \cdot \nabla(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) - r_h^{(n),\theta}(\theta u_h^{(n)} + (1-\theta) u_h^{(n-1)}) \Big] \Big|_{\widetilde{K}}$$

for all elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ in one step. We only present the computation of the values of $f_{h,\tau}$ here and refer the reader to Appendix A.4 for the remaining terms. Note that the structure Options.Dependence stores the dependencies of the data function, in particular for f, which allows to distinguish specific cases for the data.

```
0200        QuadraturVertices = ...
0201                    A+(B-A)*QNodes(j,1)+(C-A)*QNodes(j,2);
0202
0203        %Collecting f values
0204        FhTerm = zeros(Nnze,2);
0205        FTerm = zeros(Nnze,2);
0206        if Options.Dependence.F<1
0207           FhTerm = ones(Nnze,2)*ValF;
0208           FTerm = FhTerm;
0209           FhTheta = FhTerm(:,1);
0210        else
0211           switch (Options.Dependence.F)
0212               case 1
0213                  FhTerm(:,2) = ones(Nnze,1)*ValF(2);
0214                  FTerm(:,2) = FhTerm(:,2);
0215                  if BackwardValue
0216                      FhTerm(:,1) = ones(Nnze,1);
0217                      FTerm(:,1) = FhTerm(:,1);
0218                      FhTheta = Theta*FhTerm(:,2) + (1-Theta)*FhTerm(:,2);
0219                  else
0220                      FhTheta = FhTerm(:,2);
0221                  end
0222               case 2
0223                  FhTerm(:,2) = QBasisfct(j,:)*...
0224                                              [Projection.F(Nodes(:,1),1),...
0225                                               Projection.F(Nodes(:,2),1),...
0226                                               Projection.F(Nodes(:,3),1)]';
0227                  FTerm(:,2) = feval(Driver,'f(x,t)',QuadraturVertices);
0228                  FhTheta = FhTerm(:,2);
0229               case 3
0230                  FhTerm(:,2) = QBasisfct(j,:)*...
0231                                              [Projection.F(Nodes(:,1),2),...
0232                                               Projection.F(Nodes(:,2),2),...
```

```
                                                    Projection.F(Nodes(:,3),2)]';
                    FTerm(:,2) = feval(Driver,'f(x,t)',QuadraturVertices,T);
                    if BackwardValue
                        FhTerm(:,1) = QBasisfct(j,:)*...
                    [Projection.F(Nodes(:,1),1) Projection.F(Nodes(:,2),1),...
                                                     Projection.F(Nodes(:,3),1)]';
                        FTerm(:,1) = ...
                                feval(Driver,'f(x,t)',QuadraturVertices,Told);
                        FhTheta = Theta*FhTerm(:,2) + (1-Theta)*FhTerm(:,1);
                    else
                        FhTheta = FhTerm(:,2);
                    end
                end
            end
```

Starting with the element residuals $R_{\widetilde{K}}$, we continue with the contributions to the f-data estimator $\theta^2\|f(t_n) - f_h^{(n)}\|^2_{\widetilde{K}} + (1-\theta)^2\|f(t_{n-1}) - f_h^{(n-1)}\|^2_{\widetilde{K}}$, c.f. Equation (3.87), and the contributions to the data element residuals

$$\begin{aligned} D_{\widetilde{K}} =& \{-\operatorname{div}\{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} \\ &+ (\vec{c}_h^{(n),\theta} - \vec{c}^{(n)}) \cdot \nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)}) + (r_h^{(n),\theta} - r^{(n),\theta})(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\}|_{\widetilde{K}}. \end{aligned}$$

```
            %Collect u_h^n-u_h^n-1
            DiffU = (QBasisfct(j,:)*[U1-U1old U2-U2old U3-U3old]')';

            %Element residual
            a = FhTheta - DiffU/LocTau + DivDhTerm - ChTerm - RhTerm;
            ElementResiduals =  ElementResiduals + QWeights(j)*DoubleArea.*a.^2;

            %F-data error estimator
            a = Theta^2*(FTerm(:,2)-FhTerm(:,2)).^2;
            if BackwardValue
                a = a + (1-Theta)^2*(FTerm(:,1)-FhTerm(:,1)).^2;
            end
            FDataErrorEstimators = FDataErrorEstimators + ...
                                                    QWeights(j)*DoubleArea.*a;

            %data residuals
            a = -DivDhTerm + DivDTerm + ChTerm - CTerm + RhTerm - RTerm;
            DataResiduals = DataResiduals + QWeights(j)*DoubleArea.*a.^2;
```

Furthermore, we obtain the values for the contribution to the edge residuals

$$R_{\widetilde{E}} = \begin{cases} -[\vec{n} \cdot \{D_h^{(n),\theta}\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ g_{h,\tau} - \vec{n} \cdot \{D_h^{(n),\theta}\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D. \end{cases}$$

and to the data edge residuals $D_{\widetilde{E}}$

$$D_{\widetilde{E}} = \begin{cases} [\vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\}]_{\widetilde{E}} & \text{if } \widetilde{E} \not\subseteq \Gamma, \\ \vec{n}_{\widetilde{E}} \cdot \{(D_h^{(n),\theta} - D^{(n),\theta})\nabla(\theta u_h^{(n)} + (1-\theta)u_h^{(n-1)})\} & \text{if } \widetilde{E} \subseteq \Gamma_N, \\ 0 & \text{if } \widetilde{E} \subseteq \Gamma_D, \end{cases}$$

for all edges $\widetilde{E} \in \widetilde{\mathcal{E}}^{(n)}$. In order to identify the values of a specific edge, we use matrices. For an oriented edge E between the nodes a_E and b_E, the corresponding values can be found at the a_Eth row and b_Eth column, respectively. For the ease of presentation, we do not present the contributions of the Neumann boundary data function $g_{h,\tau}$.

```
0334   ValuesFlowh = zeros(Nnze,3);
0335   ValuesFlow = zeros(Nnze,3);
0336   for i=1:3
0337   %Collecting D values
0338       if Options.Dependence.D == 0
0339            DhEdge = ValD;
0340            DEdge =  ValD;
0341            ValuesFlowh(:,i) = ...
0342              -dot(Normalvectors(:,2*i-1:2*i),(DhEdge*NablaUTheta')',2);
0343            ValuesFlow(:,i) = ...
0344               -dot(Normalvectors(:,2*i-1:2*i),(DEdge*NablaUTheta')',2);
0345       elseif Options.Dependence.D == 1
0346             DhEdge = Theta*ValD(3:4,:) + (1-Theta)*ValD(1:2,:);
0347             DEdge =  DhEdge;
0348             ValuesFlowh(:,i) = ...
0349              -dot(Normalvectors(:,2*i-1:2*i),(DhEdge*NablaUTheta')',2);
0350             ValuesFlow(:,i) = ...
0351               -dot(Normalvectors(:,2*i-1:2*i),(DEdge*NablaUTheta')',2);
0352       else
0353             DhEdge = (Projection.D(Nodes(:,i),:,:)+...
0354                       Projection.D(Nodes(:,mod(i,3)+1),:,:))/2;
0355             DhEdge = reshape(permute(DhEdge,[2 1 3]),2*size(A,1),2);
0356             DhEdge = blktridiag(DhEdge,size(A,1));
0357             DEdge = Theta*feval(Driver,'D(x,t)',...
0358                         (CommonGrid.C4N(Nodes(:,i),:)+...
0359                          CommonGrid.C4N(Nodes(:,mod(i,3)+1),:))/2,T);
0360             if BackwardValue
0361                   DEdge = DEdge + (1-Theta)*feval(Driver,'D(x,t)',...
0362                         (CommonGrid.C4N(Nodes(:,i),:)+...
0363                          CommonGrid.C4N(Nodes(:,mod(i,3)+1),:))/2,T);
0364             end
0365             DEdge = reshape(permute(DEdge,[2 1 3]),2*size(A,1),2);
0366             DEdge = blktridiag(DEdge,size(A,1));
0367             ValuesFlowh(:,i) = -dot(Normalvectors(:,2*i-1:2*i),...
0368                  reshape(DhEdge*reshape(NablaUTheta',2*size(A,1),1),...
0369                                                  2,size(A,1))',2);
0370             ValuesFlow(:,i) = -dot(Normalvectors(:,2*i-1:2*i),...
0371                  reshape(DEdge*reshape(NablaUTheta',2*size(A,1),1),...
0372                                                  2,size(A,1))',2);
0373       end
0374
0375    end
0376    ValuesLength = reshape(EdgeLengths',3*Nnze,1);
0377    ValuesFlowh = reshape(ValuesFlowh',3*Nnze,1);
0378    ValuesFlow = reshape(ValuesFlow',3*Nnze,1);
```

```
0379
0380   %Assembly of flow matrices
0381    Flowh = ...
0382    sparse(I1,I2,ValuesFlowh,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0383    Flow = ...
0384     sparse(I1,I2,ValuesFlow,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0385    Length = ...
0386   sparse(I1,I2,ValuesLength,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));

0394   %Data edge residuals
0395   Integrand = (Flowh - Flow + Flowh' - Flow' ).^2;
0396   DataEdgeResidual = Length .* Integrand;
0476    %Edge residuals
0477    Integrand = (Flowh + Flowh').^2;
0478    EdgeResidual = Length .* Integrand;
```

We did not consider the local parameters $\alpha_{\widetilde{K}}$ yet. To obtain the element terms

$$\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|R_{\widetilde{K}}\|_{\widetilde{K}}^2$$

of the Verfürth-type error estimator $\eta_h^{(n)}$,

$$\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \|D_{\widetilde{K}}\|_{\widetilde{K}}^2,$$

of the data error estimator $\Theta_h^{(n)}$, and

$$\sum_{\widetilde{K}\in\widetilde{\mathcal{T}}^{(n)}} \alpha_{\widetilde{K}}^2 \Big[\theta^2\|f(t_n) - f_h^{(n)}\|_{\widetilde{K}}^2 + (1-\theta)^2\|f(t_{n-1}) - f_h^{(n-1)}\|_{\widetilde{K}}^2\Big]$$

of the f-data error estimator, we have to multiply the $\widetilde{K}$-local contributions with $\alpha_{\widetilde{K}}^2$.

```
0489    Diameters = max(EdgeLengths,[],2);
0490    if Beta
0491        AlphaK = min([Diameters*Lambda^(-1/2),Beta^(-1/2)*ones(Nnze,1)],[],2);
0492    else
0493        AlphaK = Diameters*Lambda^(-1/2);
0494    end
0495    VerfurthErrorEstimator = sum(AlphaK.^2.*ElementResiduals);
0496    FDataErrorEstimator = sum(AlphaK.^2.*FDataErrorEstimators);
0497    DataErrorEstimator = sum(AlphaK.^2.*DataResiduals);
0498    RefinementIndicator = AlphaK.^2.*...
0499                        (ElementResiduals+DataResiduals+FDataErrorEstimators);
```

Note that we also considered the element contributions to the refinement indicator

$$\alpha_{\widetilde{K}}^2\{\|R_{\widetilde{K}}\|_{\widetilde{K}}^2 + \|D_{\widetilde{K}}\|_{\widetilde{K}}^2 + \theta^2\|f(t_n) - f_h^{(n)}\|_{\widetilde{K}}^2 + (1-\theta)^2\|f(t_{n-1}) - f_h^{(n-1)}\|_{\widetilde{K}}^2\}$$

in the previous code passage. For the edge contributions to the quantities above, we again use variables of sparse matrix type.

```
0501    %edges
0502    %Obtain AlphaE
0503    ValAlphaE = zeros(Nnze,3);
0504    for i=1:3
0505       if Beta
0506          ValAlphaE(:,i) = min([EdgeLengths(:,i)*Lambda^(-1/2),...
0507                                              Beta^(-1/2)*ones(Nnze,1)],[],2);
0508       else
0509          ValAlphaE(:,i) = EdgeLengths(:,i)*Lambda^(-1/2);
0510       end
0511    end
0512    ValAlphaE = reshape(ValAlphaE',3*Nnze,1);
0513    AlphaE = ...
0514      sparse(I1,I2,ValAlphaE,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0515
0516    LocEdgeContribution = Lambda^(-1/2)*AlphaE.*...
0517                 (EdgeResidual + DataEdgeResidual + GContribution);
0518    GContributionLocal = Lambda^(-1/2)*sum(sum(AlphaE.*GContribution));
```

Finally, the edge contribution to the refinement indicators

$$\sum_{i=1}^{3} \lambda^{-1/2}\alpha_{\widetilde{E}_i}\{\|R_{\widetilde{E}_i}\|^2_{\widetilde{E}_i} + \|D_{\widetilde{E}_i}\|^2_{\widetilde{E}_i} + \theta^2\|g(t_n) - g_h^{(n)}\|^2_{\widetilde{E}_i} + (1-\theta)^2\|g(t_{n-1}) - g_h^{(n-1)}\|^2_{\widetilde{E}_i}\}$$

and the edge contributions to the Verfürth-type error estimator $\eta_h^{(n)}$ and to the data error estimator $\Theta_h^{(n)}$

$$\sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|R_{\widetilde{E}}\|^2_{\widetilde{E}}, \qquad \sum_{\widetilde{E}\in\widetilde{\mathcal{E}}^{(n)}} \lambda^{-1/2}\alpha_{\widetilde{E}}\|D_{\widetilde{E}}\|^2_{\widetilde{E}},$$

are computed.

```
0520    %refinement indicators
0521    RefinementIndicator = RefinementIndicator + ...
0522                extractValues(LocEdgeContribution,Nodes(:,1),Nodes(:,2))+...
0523                extractValues(LocEdgeContribution,Nodes(:,2),Nodes(:,3))+...
0524                extractValues(LocEdgeContribution,Nodes(:,3),Nodes(:,1));
0525
0526    %Verfuerth-type error indicator
0527    VerfurthErrorEstimator = VerfurthErrorEstimator ...
0528                + Lambda^(-1/2)*sum(sum(AlphaE.*EdgeResidual.*EdgeCounter));
0529    DataErrorEstimator = DataErrorEstimator ...
0530            + Lambda^(-1/2)*sum(sum(AlphaE.*DataEdgeResidual.*EdgeCounter));
```

The entire code can be found in Appendix A.4.

3.7.2 The Function *etaTemporal*

This function computes the temporal error estimator $\eta_\tau^{(n)}$, c.f. (3.93). We first call the subfunction *getCommonU* to obtain the values of $u_{h,\tau}(t_{n-1})$ and $u_{h,\tau}(t_n)$ at the vertices of the common

refinement $\widetilde{\mathcal{T}}^{(n)}$ by linear interpolation. Independently of the data functions, we have to compute the term $\|\!|u_h^n - u_h^{(n-1)}|\!\|_{\widetilde{K}}$ for all elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$. In the following code the variables $N1$, $N2$ and $N3$ denote the $\widetilde{\mathcal{T}}^{(n)}$-constant gradients of the nodal basis functions corresponding to the vertices of $\widetilde{K}$.

```
0089   %Common U values
0090   U1 = UCommon(CommonGrid.N4E(nze,1),1);
0091   U2 = UCommon(CommonGrid.N4E(nze,2),1);
0092   U3 = UCommon(CommonGrid.N4E(nze,3),1);
0093   U1old = UCommon(CommonGrid.N4E(nze,1),2);
0094   U2old = UCommon(CommonGrid.N4E(nze,2),2);
0095   U3old = UCommon(CommonGrid.N4E(nze,3),2);
0096
0097   %Assembly of gradient term of energy norm
0098   a = ((U1-U1old)*ones(1,2)).*N1 ...
0099        + ((U2-U2old)*ones(1,2)).*N2 + ((U3-U3old)*ones(1,2)).*N3;
0100   sum1 = sum((a(:,1).^2+a(:,2).^2).*CommonGrid.DoubleArea)/2;
0101
0102   %Assembly of L2 term if energy norm
0103   sum2 = 0;
0104   for i=1:size(QWeights,1)
0105      sum2 = sum2 + QWeights(i)*...
0106             sum(CommonGrid.DoubleArea'.*(Options.QBasisfct(i,:)*...
0107                                    [(U1-U1old) (U2-U2old) (U3-U3old)]').^2);
0108   end
```

For the contributions including time-derivatives of the data functions

$$(1+\theta)^2\tau_n\Bigg[\int_{t_{n-1}}^{t_n}\|\dot{f}(t)\|^2dt+\int_{t_{n-1}}^{t_n}\|\dot{g}(t)\|^2_{\Gamma_N}dt+\lambda^{-1}\int_{t_{n-1}}^{t_n}\|\dot{D}(t)\nabla u_{h,\tau}\|^2dt$$
$$+\int_{t_{n-1}}^{t_n}\|\dot{\vec{c}}(t)\cdot\nabla u_{h,\tau}\|^2dt+\int_{t_{n-1}}^{t_n}\|\dot{r}(t)u_{h,\tau}(\overline{t}_n)\|^2dt\Bigg]\Bigg\}^{1/2}$$

we only present the term including $\dot{f}$. We refer the reader to Appendix A.5 for the respective codes for the other terms. Generally, we consider specific cases of spatially independent data functions. For time integration, we use the third order midpoint rule.

```
0110   %Consider space-time dependent f-data
0111   if Options.Dependence.F==3
0112       for i=1:size(QWeights,1)
0113           QuadraturVertices = ...
0114                   A + (B-A)*Options.QNodes(i,1)+(C-A)*Options.QNodes(i,2);
0115           EtaTemporal2 = EtaTemporal2 + ...
0116                  QWeights(i)*sum(CommonGrid.DoubleArea.*...
0117                   (feval(Driver,'df(x,t)/dt',QuadraturVertices,MeanT)).^2);
0118       end
0119
0120   elseif Options.Dependence.F==1
0121
0122       %spatially constant rhs
```

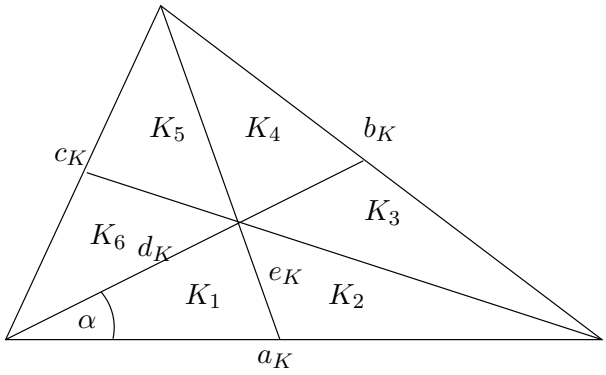


Figure 3.4: Orange refinement

```
0123        EtaTemporal2 = sum(CommonGrid.DoubleArea(CommonElement))/2*...
0124                                   (feval(Driver,'df(x,t)/dt',[0 0],MeanT))^2;
0125    end
```

The entire code can be found in Appendix A.5.

3.8 A Common Refinement Triangulation

Due to the considerations of the previous sections, we need a regular triangulation $\widetilde{\mathcal{T}}$, which is a common refinement of two given regular triangulations from adjacent time steps, i.e. $\mathcal{T}^{(n-1)}$ and $\mathcal{T}^{(n)}$, with fixed $0 \leq n \leq N$. To decrease the complexity of the algorithm we assume:

CONVENTION. Let $\mathcal{T}^{(n-1)} =: \overline{\mathcal{G}}^{(n-1)}$, $\mathcal{T}^{(n)} =: \overline{\mathcal{G}}^{(n)}$ be the regular triangulations that correspond to two adjacent time steps. Then, the grid $\mathcal{G}^{(n)}$ can be obtained from $\mathcal{G}^{(n-1)}$ by at most one red-coarsening and $k \in \mathbb{N}$ successive red-refinements.

Thus, in the computation of $\widetilde{\mathcal{T}}$ it pays to start with $\mathcal{T}^{(n)}$ as default, since usually this will be the finer triangulation. This approach gives eight general cases for the elements of $\mathcal{T}^{(n)}$, c.f. Table 3.1. Note that we denote an element as "white element", if it is no green brother. Due to the cases 6 and 8, we need an additional refinement technique.

CONVENTION. To obtain the triangulation $\widetilde{\mathcal{T}}$, we additionally allow the following refinement:

(i) **Orange-refinement:** The element $K \in \mathcal{T}$ is split into 6 triangles $K_1, \ldots, K_6 \in \mathcal{T}$, such that $K_1 \cup K_2 \cup K_3 \cup K_4 \cup K_5 \cup K_6 = K$, c.f. Figure 3.4. Additional to the original vertices of K, the vertices of K_i, $i = 1, \ldots, 6$, include the three barycenters of the edges of K as well as the barycenter of K itself. Here, the exact affiliation of the nodes is according to case 6 in Table 3.1.

Lemma 3.24. *For the triangles K_1,...,K_6 obtained from orange-refinement of $K \in \mathcal{T}$, there holds $h_{K_i} \leq 2h_K/3$ and $|K_i| = |K|/6$ for $i = 1, \ldots, 6$. In particular $h_{K_i}^2/|K_i| \leq 8h_K^2/(3|K|)$, i.e. the shape regularity constant increases at most by a factor $8/3$.*

Proof. Consider the notations from Figure 3.4. Lemma 1.1 implies $|K|/2 = a_K d_K \sin\alpha/2$.

Table 3.1: Possible common refinements for elements from two adjacent time steps.

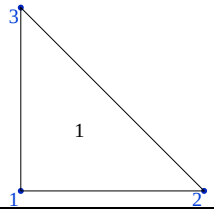
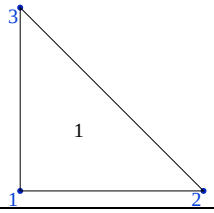
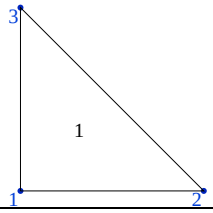
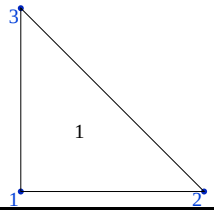
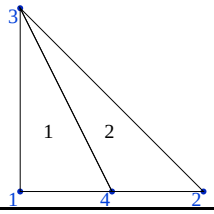
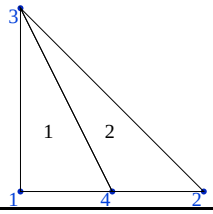
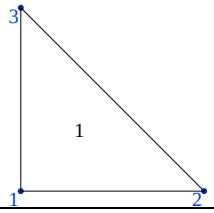
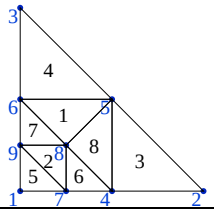
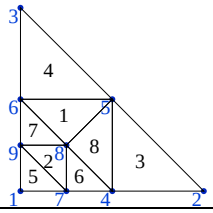
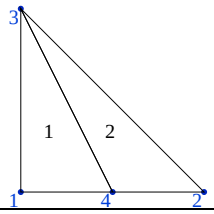
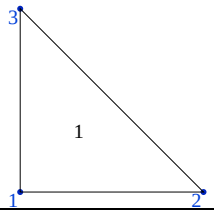
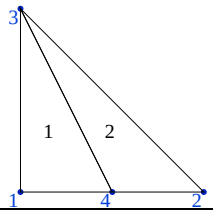
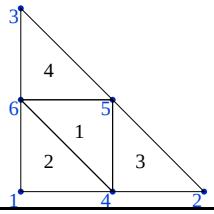
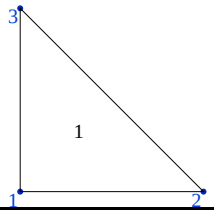
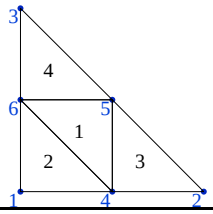
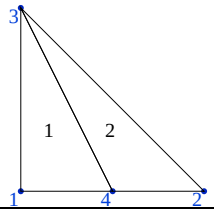
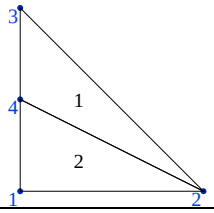
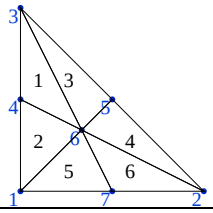
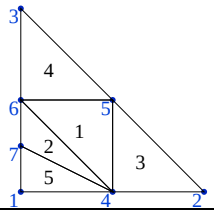
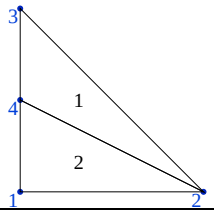
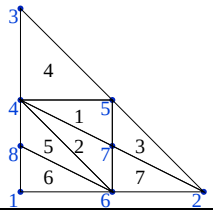
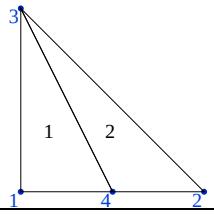
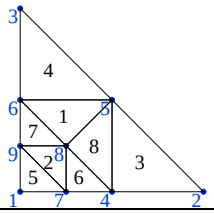
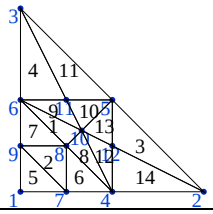
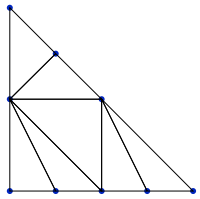
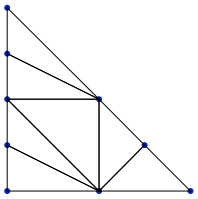
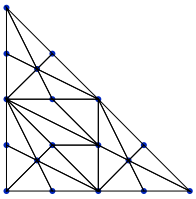
Case	Element from $\mathcal{T}^{(n-1)}$	Element from $\mathcal{T}^{(n)}$	Common refinement	Comments
1				No change.
2				The white element has been green-refined.
3				The white element has been red-refined. Multiple refinements are possible.
4				The twin element has become white.
5				The element has been red-coarsened.
6				The twin element changed to a different version.
7				The element has been red-coarsened and then green refined.
8				The twin element has been red-refined. Multiple refinements are possible.

Table 3.2: The worst case scenario in terms of hanging nodes, due to orange refinement. Here, K denotes the middle element of $\mathcal{T}^{(n)}$ and $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ any of the elements, such that $\widetilde{K} \subset K$.

Case	Element from $\mathcal{T}^{(n-1)}$	Element from $\mathcal{T}^{(n)}$	Common refinement	Comments
6b				There holds $h^2_{\widetilde{K}}/\lvert\widetilde{K}\rvert \leq 4h^2_K/\lvert K\rvert$.

The barycenter splits d_K with the ratio $2:1$, and so

$$|K_1| = \frac{1}{2}\Big(\frac{a_K}{2}\frac{2d_K}{3}\Big)\sin\alpha = \frac{|K|}{6}.$$

Consequently, we have $h_{K_1} = \max\{a_K/2, 2d_K/3, e_K/3\} \leq 2\max\{a_K, d_K, e_K\}/3 \leq 2h_K/3$. Analogous considerations for i=2,...,5 prove the lemma. ■

Since orange refinement may create two new hanging nodes for an element, considering solely the eight cases from Table 3.1 generally violates the grid condition. To obtain a regular triangulation it is therefore necessary to conduct a grid and green closure, respectively. The grid closure introduced in Section 1.2 is not practicable in this context since red-refinement of all elements with more than one hanging node, may not result in a common refinement. Thus, we choose a double green-refinement strategy, c.f. Case 6*b* Table 3.2. Let K be the middle element of the $\mathcal{T}^{(n)}$-column and $\widetilde{K}$ one of the resulting double green-refined triangles. Using (1.1) twice, we then obtain an estimate for the shape regularity for the case 6*b*

$$\frac{h^2_{\widetilde{K}}}{|\widetilde{K}|} \leq \frac{4h^2_K}{|K|}. \tag{3.97}$$

Lemma 3.25. *Let $c_{S,0} := \max_{K\in\mathcal{T}^{(0)}} h_K/\rho_K$, i.e. $c_{S,0}$ is the shape regularity constant of the initial triangulation. Then, there holds*

$$c_S \leq 2c_{S,0} \tag{3.98}$$
$$c^{(n)}_{\widetilde{S}} \leq 4c_{S,0} \tag{3.99}$$
$$c_T \leq 8c_{S,0} \tag{3.100}$$

for all n $1 \leq n \leq N$. Here, $c^{(n)}_{\widetilde{S}}$ denotes the shape regularity constant of the common refinements $\widetilde{\mathcal{T}}^{(n)}$.

Proof. The first estimate is another way stating (1.1). For the second inequality we focus on Table 3.1, denoting the elements of $\widetilde{\mathcal{T}}^{(n)}$ according to case and number in the table, e.g. $\widetilde{K}_{3,8}$ is the 8^{th} element of case 3 in the fourth column. Analogously, $K_{i,j}$ denotes the j^{th} element of the i^{th} case of $\mathcal{T}^{(n)}$.

(i) **Cases 1–5:** Since there is no further refinement of the elements of $\mathcal{T}^{(n-1)}$ or $\mathcal{T}^{(n)}$ there holds

$$\mu_{\widetilde{K}_{i,j}} \leq 2c_{S,0}$$

according to (3.98), where $i = 1, \dots, 5$ and $j = 1, \dots, 8$ denotes any element of these cases.

(ii) **Cases 6 & 8:** Here, Lemma 3.24 gives the upper bound

$$\mu_{\widetilde{K}_{i,j}} \leq \frac{8}{3} c_{S,0}$$

where $i = 6, 8$ and $j = 1, \dots, 13$ denotes any element of these cases.

(iii) **Case 7:** Note, that the elements $\widetilde{K}_{7,i}$ $i = 1, 2, 3, 5, 6, 7$ are similar to either $K_{7,1}$ or $K_{7,2}$. Thus, shape regularity does not change with respect to c_S

$$\mu_{\widetilde{K}_{7,j}} \leq 2 c_{S,0}$$

where $j = 1, \dots, 7$ denotes any element of this case.

Still, we have to take into account the considerations that led to (3.97), which yields the maximum bound for $c_{\widetilde{S}}^{(n)}$, proving (3.99). To prove the last estimate we note that

$$\frac{h_K}{h_{\widetilde{K}}} = \frac{h_K \rho_{\widetilde{K}}}{|\widetilde{K}|} \leq \frac{h_K^2}{|\widetilde{K}|} \leq c_S \frac{|K|}{|\widetilde{K}|} \tag{3.101}$$

for all elements $\widetilde{K} \in \widetilde{\mathcal{T}}^{(n)}$ with $\widetilde{K} \subseteq K \in \mathcal{T}^{(n)}$. With this inequality, Lemma 1.1, and Table 3.1 we obtain individual estimates for each case

(i) **Cases 1–3:** There is no refinement of K, thus the ratios $h_K / h_{\widetilde{K}}$ equal 1.

(ii) **Case 4:** There holds $h_K / h_{\widetilde{K}} \leq 2 c_S$.

(iii) **Case 5:** There holds $h_K / h_{\widetilde{K}} \leq 4 c_S$.

(iv) **Case 6:** There holds $h_K / h_{\widetilde{K}} \leq 3 c_S$.

(v) **Case 7:** There holds $h_K / h_{\widetilde{K}} \leq 4 c_S$.

(vi) **Case 8:** There holds $h_K / h_{\widetilde{K}} \leq 3 c_S$.

Due to the regularization of $\widetilde{\mathcal{T}}^{(n)}$, c.f. Table 3.2, we have one additional case, where $h_K / h_{\widetilde{K}} \leq 4 c_S$. Thus the maximum of these estimates and (3.98) prove (3.100). ■

In particular, for the used grid management strategy the previous lemma proves that the shape regularity constant c_S and the transition constant c_T, which show up in most of the decisive estimates in the previous sections, are bounded by a constant which is predefined by the given initial triangulation.

3.8.1 The Function *commonGrid*

In the following, we denote the triangulations $\mathcal{T}^{(n-1)}$, $\mathcal{T}^{(n-1)}$, $\widetilde{\mathcal{T}}$ by their computational counterparts *OldGrid*,*Grid* and *CommonGrid*. Generally, as *Grid* may derive from *OldGrid* by a combination of one red-coarsening and multiple red-refinements, relating elements from *Grid* to elements from *OldGrid* is rather complicate. Due to the reuse of rows in *Grid.C4N*, *Grid.N4E* and *Grid.F4E*, c.f. Section 1.3, entries of the coordinates, the element and its father may simply be trivial in *OldGrid* or filled with non corresponding data. Moreover, the correct element in *OldGrid* to compare to, may be shifted anywhere in the storage. Thus, we use the condition

that there is only one step of red-coarsening. Recall, that the middle child in red-refinement is stored at the row of the original element. Hence, we may always find a corresponding element in *OldGrid*, by searching *Grid.F4E* for the correct middle element. Fortunately, due to the nested structure it is not necessary to check all middle child entries of *Grid.F4E*.
As initial value for *CommonGrid*, we set *Grid*, since this is expected to be the finer triangulation. We intend to store the element in which a triangle of *CommonGrid* is contained in the fourth and fifth column of *CommonGrid.N4E* for *Grid* and *OldGrid*, respectively. First, we check whether the two grids *OldGrid* and *Grid* are the same. If so, we can stop the computation, since *Grid* is already a common refinement.

```
0104   %Check if Grid and OldGrid are the same
0105   nze = setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E);
0106   if isequal(OldGrid.C4N,Grid.C4N) && isequal(OldGrid.N4E,Grid.N4E)
0107       GridsAreEqual = 1;
0108       CommonGrid = Grid;
0109       CommonGrid.N4E(nze,4:5) = [nze' nze'];
0110       return;
0111   else
0112       GridsAreEqual = 0;
0113   end
```

To further decrease the elements, for which we have to find a common refinement, we directly compare *OldGrid.N4E* to *Grid.N4E* and *OldGrid.C4N* to *Grid.C4N*. For most of the unchanged elements this identification method works, so we reduce the elements in the upcoming loop.

```
0117   %default value of CommonGrid is Grid
0118   CommonGrid = Grid;
0119   %prelocation
0120   Ngb = nnz(find(Grid.N4E(:,5))); %number of green brothers
0121   %all other elements
0122   Nngb = max(size(Grid.N4E,1)-nnz(Grid.Unused.N4E)- Ngb,0);
0123   CommonGrid.ElementList = zeros(Ngb*2+Nngb*3,5);
0124   CommonGrid.ElementCount = 0;
0125   CommonGrid.VertexCount = size(CommonGrid.C4N,1);
0126   CommonGrid.C4N = [CommonGrid.C4N; zeros(Ngb*2+Nngb*3,2)];
0127
0128   CommonGrid.N4E = [CommonGrid.N4E(:,1:3) zeros(size(CommonGrid.N4E,1),2)];
0129   a = CommonGrid.N4E(nze,1:3)';
0130   I1 = reshape(a,3*nnz(nze),1);
0131   a = CommonGrid.N4E(nze,[2 3 1])';
0132   I2 = reshape(a,3*nnz(nze),1);
0133   CommonGrid.NOE = CommonGrid.NOE ...
0134           + sparse(I1,I2,-1,size(CommonGrid.NOE,1),size(CommonGrid.NOE,1));
0135   CommonGrid.NOE = [CommonGrid.NOE sparse(size(CommonGrid.NOE,1),nnz(I1));
0136                     sparse(nnz(I1),size(CommonGrid.NOE,1)+nnz(I1))];

0138   %find some of the unchanged elements
0139   Cand = setdiff(nze,size(OldGrid.N4E,1)+1:max(nze));
0140   a = Grid.N4E(Cand,1:3)-OldGrid.N4E(Cand,1:3);
0141   a = find(a(:,1).^2 + a(:,2).^2 + a(:,3).^2==0);
```

```
0142   if ~isempty(a)
0143
0144       A = Grid.C4N(Grid.N4E(Cand(a),1),:)...
0145                                     -OldGrid.C4N(OldGrid.N4E(Cand(a),1),:);
0146       B = Grid.C4N(Grid.N4E(Cand(a),2),:)...
0147                                     -OldGrid.C4N(OldGrid.N4E(Cand(a),2),:);
0148       C = Grid.C4N(Grid.N4E(Cand(a),3),:)...
0149                                     -OldGrid.C4N(OldGrid.N4E(Cand(a),3),:);
0150       b = find(A(:,1).^2+A(:,2).^2+B(:,1).^2 ...
0151                                        +B(:,2).^2+C(:,1).^2+C(:,2).^2==0);
0152       Unchanged = Cand(a(b));
0153       CommonGrid.N4E(Unchanged,4:5) = [Unchanged' Unchanged'];
0154       nze = setdiff(nze,Unchanged);
0155   end
```

In a loop over the remaining non trivial entries of $Grid.N4E$, we now try to identify the elements one by one. At the lines (198)–(361), we carry out this identification, i.e. the process to find the right element(s) in *OldGrid* to compare to. Then, we decide which specific case is to be treated (see Table 3.1 for the specification of these cases).

```
0378       %Set identifier
0379       NotIdentified = 1; %prevent multiple checks for performance
0380       if OldElementGreenBrother
0381          %default1: green->red(multiple)
0382          CommonIdentifier = 8;
0383       else
0384          %default2: white->red(multiple)
0385          CommonIdentifier = 3;
0386       end
0387
0388       if isequal(OldVertices,Vertices) || (OldElementGreenBrother  ...
0389                          && isequal(OldElementGreenBrotherVertices,Vertices))
0390           %no change
0391           CommonIdentifier = 1;
0392           NotIdentified = 0;
0393       end
0394
0395       if  NotIdentified && (ElementGreenBrother ...
0396             && ~OldElementGreenBrother && isequal(OldVertices,TwinVertices))
0397           %white->green
0398           CommonIdentifier = 2;
0399           NotIdentified = 0;
0400       end
0401
0402       if  NotIdentified && ~ElementGreenBrother ...
0403               && OldElementGreenBrother && isequal(OldTwinVertices,Vertices)
0404           %green->white
0405           CommonIdentifier = 4;
0406           NotIdentified = 0;
0407       end
```

```
0408
0409      if  NotIdentified && ...
0410                          (OldFather && isequal(OldFatherVertices,Vertices))
0411          %red(+green)->white
0412          CommonIdentifier = 5;
0413          NotIdentified = 0;
0414      end
0415
0416      if  NotIdentified && ElementGreenBrother ...
0417          && OldElementGreenBrother ...
0418          && isequal(OldTwinVertices,TwinVertices)...
0419          && ~isequal(OldTwinVertices,Vertices)
0420          %green->green(different shape)
0421          CommonIdentifier = 6;
0422          NotIdentified = 0;
0423      end
0424
0425      if  NotIdentified && OldFather && ElementGreenBrother && ...
0426             isequal(OldFatherVertices,TwinVertices)
0427           %Element was red(+green) and changed to green
0428           CommonIdentifier = 7;
0429      end
```

Note, that Case 1 also includes twins which remained unchanged during the refinement and coarsening processes. Finally, we compute the common refinement for the current element or we call subfunctions for the more complicated Cases 4–8. The subfunction *commonGrid*8, which considers Case 8, has to be recursive, since the number of red-refinements between the identified element in *OldGrid* and the current element in *Grid* is initially unknown.

```
0431      %Compute local contribution to CommonGrid
0432      switch (CommonIdentifier)
0433
0434           case 1
0435               %no change
0436               if OldElementGreenBrother && ...
0437                           isequal(OldElementGreenBrotherVertices,Vertices)
0438                  CommonGrid.N4E(Element,4:5) = ...
0439                                          [Element OldElementGreenBrother];
0440               else
0441                  CommonGrid.N4E(Element,4:5) = [Element OldElement];
0442               end
0443
0444           case 2
0445               %white->green
0446               CommonGrid.N4E(Element,4:5) = [Element OldElement];
0447
0448           case 3
0449               %white->red(multiple)
0450               CommonGrid.N4E(Element,4:5) = [Element OldElement];
0451
```

```
0452            case 4
0453                %green->white
0454                CommonGrid = ...
0455                          commonGrid4(Element,OldElement,OldGrid,CommonGrid);
0456
0457            case 5
0458                %red(+green)->white
0459                CommonGrid = ...
0460                          commonGrid5(Element,OldElement,OldGrid,CommonGrid);
0461
0462            case 6
0463                %green->green(different shape)
0464                CommonGrid = commonGrid6(Element,OldElement,...
0465                                    ElementGreenBrother,OldGrid,CommonGrid);
0466
0467            case 7
0468                %red(+green)->green
0469                CommonGrid = commonGrid7(Element,ElementGreenBrother,...
0470                                                OldElement,OldGrid,CommonGrid);
0471
0472            case 8
0473                %green->red(multiple)
0474                OldTwinVertices =[OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:);
0475                       OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)];
0476                CommonGrid = commonGrid8(Element,1,RelatedFathers,...
0477                   OldElement,OldTwinVertices,[1 2],OldGrid,Grid,CommonGrid);
0478
0479        end
```

The complete code of *commonGrid* can be found in Appendix A.6.

Chapter 4

Numerical Experiments

The analysis and the implementations developed in the preceding chapters are now applied to solve certain linear parabolic problems numerically. We start with a simple model problem with constant coefficients to verify the code and illustrate the convergence rate predicted by Theorem 2.6. The adaptive algorithm discussed at the end of the previous chapter is compared with a strategy with uniform time step sizes and uniform spatial mesh-width. Second, we investigate a similar example with radial symmetry. Third, we show numerical results for the heat equation. Finally, a model for pollution in ground water flow is presented[1].
Throughout, the error is measured in the energy norm of the problem, as defined in Equation (2.1), and the required tolerance is denoted by ε. For uniformly refined meshes, the error $e := |||u(T) - u_{h,\tau}(T)|||$ at final time $T > 0$ and the corresponding error estimator η are computed for a certain number of free nodes, denoted by $|\mathcal{N}_f|$. For adaptive computations, the respective quantity is the mean number of free nodes $\overline{|\mathcal{N}_f|}$ of the sequence of triangulations $\left(\mathcal{T}^{(n)}\right)$. The latter quantity is defined by

$$\overline{|\mathcal{N}_f|} := \operatorname{round}\left(\frac{1}{N+1}\sum_{n=0}^{N}|\mathcal{N}_f^{(n)}|\right), \tag{4.1}$$

where N denotes the number of time steps. Note that $\overline{|\mathcal{N}_f|} = |\mathcal{N}_f|$ if the spatial mesh is kept constant in time.
In this chapter, we use three strategies to obtain the discrete solution $u_{h,\tau}$:

Semi-uniform strategy: Either time step size τ or spatial mesh-width h are predefined. For the other quantity, we use a sequence of uniformly refined meshes.

Uniform strategy with parameter σ: Either time step size τ or uniform spatial mesh-width h are predefined. The other quantity is chosen according to $h = \tau^\sigma$ with a parameter $\sigma > 0$.

Adaptive strategy with tolerance ε: The adaptive algorithm for this strategy is introduced in Section 3.7. For each time step, the time step size τ_n and the local mesh-width function $h_{\mathcal{T}^{(n)}}$ are accordingly adapted with respect to the tolerance ε.

In the model problems below we compare uniform and adaptive strategy. To measure the quality of a strategy we use the absolute values of error and error estimator as well as the experimental convergence rate. For the uniform strategy, the latter quantity is obtained as follows: For a fixed uniform time step size τ, we analytically expect a dependence $e_h = \mathcal{O}(h^\alpha) \sim \mathcal{O}(|\mathcal{N}_f|^{-\alpha/2})$

[1] In the following, vectors in $\mathbb{R}^2$ are denoted by $\vec{x} = (x, y)$.

between the uniform mesh width h and respective error e_h according to Theorem 2.6. In this context α denotes the convergence rate. To obtain the experimental convergence rate α_e we use the ansatz $e_1 = ch_1^{\alpha_e}$ and $e_2 = ch_2^{\alpha_e}$, for two mesh-widths h_1 and h_2 with corresponding errors e_1 and e_2, respectively, and unknown constants α_e and c. This implies

$$\left(\frac{h_1}{h_2}\right)^{\alpha_e} = \frac{e_1}{e_2} \quad \text{and thus} \quad \alpha_e = \frac{\ln[e_1/e_2]}{\ln[h_1/h_2]}.$$

Written in terms of number of free nodes $|\mathcal{N}_f|$, this reads

$$\alpha_e(|\mathcal{N}_{f,1}|, |\mathcal{N}_{f,2}|) := \frac{\ln[e(|\mathcal{N}_{f,1}|)/e(|\mathcal{N}_{f,2}|)]}{\ln[\{|\mathcal{N}_{f,2}|\}^{1/2}/\{|\mathcal{N}_{f,1}|\}^{1/2}]}, \tag{4.2}$$

where the dependencies of the entities are written explicitly. We compute α_e for two adjacent solutions: If $\mathcal{T}_2$ is obtained from $\mathcal{T}_1$ by uniform red-refinement, $|\mathcal{N}_{f,1}|$ and $|\mathcal{N}_{f,2}|$ are the number of free nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. An analogous approach for the error estimator η leads to

$$\alpha_\eta(|\mathcal{N}_{f,1}|, |\mathcal{N}_{f,2}|) := \frac{\ln[\eta(|\mathcal{N}_{f,1}|)/\eta(|\mathcal{N}_{f,2}|)]}{\ln[\{|\mathcal{N}_{f,2}|\}^{1/2}/\{|\mathcal{N}_{f,1}|\}^{1/2}]}. \tag{4.3}$$

For adaptive computations, the error e as well as the error estimator η and the mean number of free nodes $\overline{|\mathcal{N}_f|}$ is an output of the adaptive algorithm for a given tolerance $\varepsilon > 0$. In the following formulae, this dependence is written explicitly, e.g. $e = e(\varepsilon)$, $\eta = \eta(\varepsilon)$. For the adaptive strategy, the experimental convergence rates provided in the tables below are always computed for two adjacent tolerances ε_1 and ε_2 with $\varepsilon_1 > \varepsilon_2$ by

$$\alpha_e(\varepsilon_1, \varepsilon_2) := \frac{\ln[e(\varepsilon_1)/e(\varepsilon_2)]}{\ln[\{\overline{|\mathcal{N}_f(\varepsilon_2)|}\}^{1/2}/\{\overline{|\mathcal{N}_f(\varepsilon_1)|}\}^{1/2}]} \tag{4.4}$$

and

$$\alpha_\eta(\varepsilon_1, \varepsilon_2) := \frac{\ln[\eta(\varepsilon_1)/\eta(\varepsilon_2)]}{\ln[\{\overline{|\mathcal{N}_f(\varepsilon_2)|}\}^{1/2}/\{\overline{|\mathcal{N}_f(\varepsilon_1)|}\}^{1/2}]} \tag{4.5}$$

for error and error estimator, respectively.

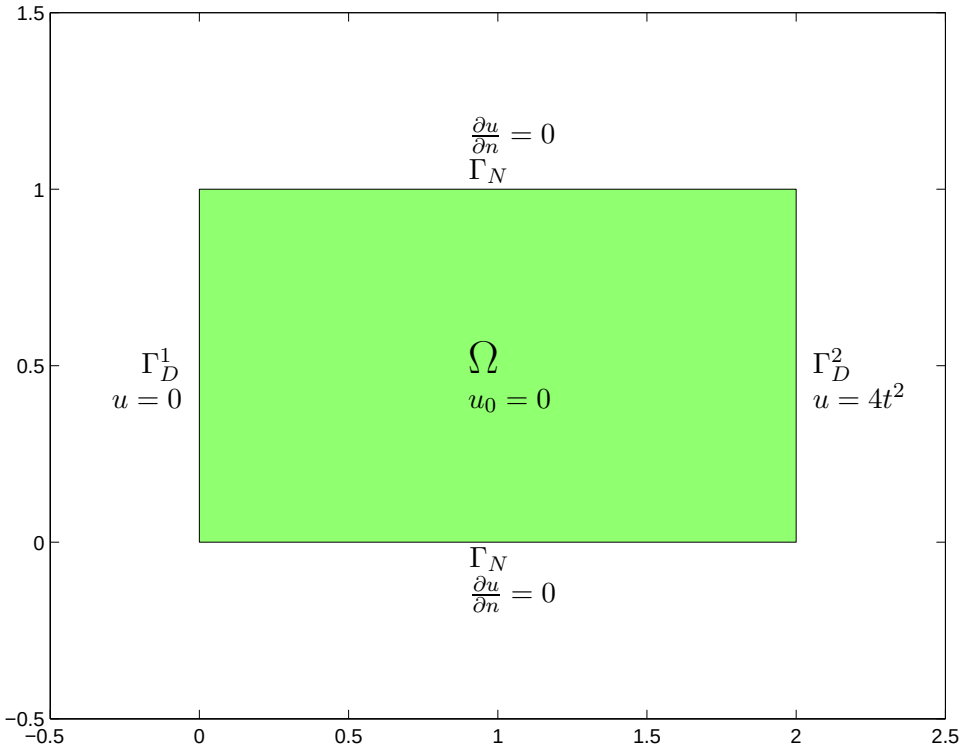


Figure 4.1: Geometry, as well as initial and boundary data in Example 4.1.

4.1 A Simple Model Problem

As a simple model problem, we consider the following linear parabolic equation with constant coefficients. We prescribe the exact solution $u(x, y, t) = t^2x^2$ and compute initial data u_0, boundary conditions, and volume force f thereof, that is, f reads $f(x, y, t) = 2tx^2 - 2t^2 + t^2x^2$.

$$
\begin{aligned}
\frac{\partial u}{\partial t} - \triangle u + u &= f \text{ in } \Omega \times (0,1], \\
u &= \begin{cases} 0 & \text{on } \Gamma_D^1 \times (0,1], \\ 4t^2 & \text{on } \Gamma_D^2 \times (0,1], \end{cases} \\
\vec{n} \cdot \nabla u &= 0 \text{ on } \Gamma_N \times (0,1], \\
u_0 &= 0 \text{ in } \Omega,
\end{aligned}
\tag{4.6}
$$

where $\Omega := (0,0) \times (2,1)$, $\Gamma_D^1 := \{0\} \times (0,1)$, $\Gamma_D^2 := \{2\} \times (0,1)$, $\Gamma_N := (0,1) \times \{0\} \cup (0,1) \times \{1\}$.

Note that $u(x, y, t)$ is, in particular, independent of y, which results in a one dimensional problem on a two dimensional domain. Figure 4.1 shows the problem's geometry. For this rectangular domain, the initial mesh for all (either uniform or adaptive) computations is shown in Figure 4.2. We choose the time discretization parameter θ to be either 1 or 1/2, which provides the backward Euler method and the Crank-Nicholson scheme, respectively. Since the curvature of the function u is constant in space, the space-time adaptive strategy leads to an almost uniformly refined triangulation at final time, c.f. Figure 4.3.

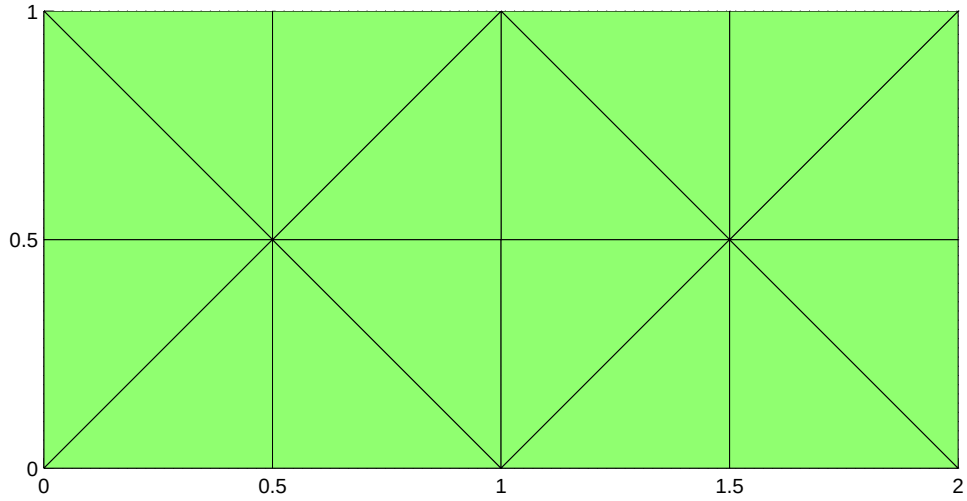


Figure 4.2: Initial triangulation in Example 4.1 with 16 elements and $|\mathcal{N}_f| = 9$ free nodes.

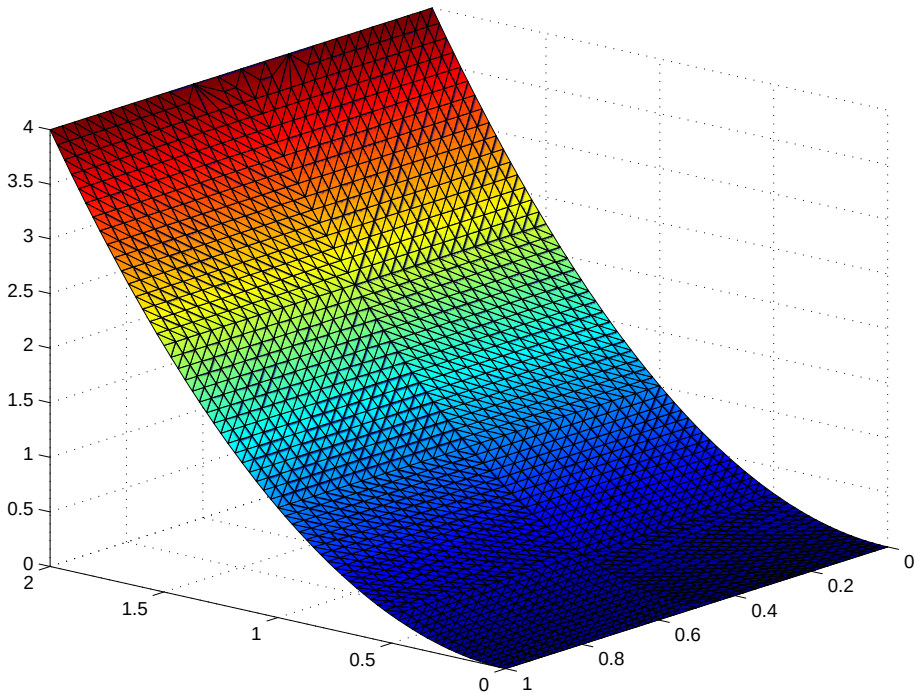


Figure 4.3: Discrete solution in Example 4.1 at final time $T = 1$ for an adaptive computation with time-discretization parameter $\theta = 1$ and tolerance $\varepsilon = 1/4$. Because of the spatially constant curvature of u, the adaptive algorithm leads to an almost uniformly refined triangulation with 4076 elements. Moreover, the adaptive algorithm used $N = 51$ time steps with a mean number of free nodes of $\overline{|\mathcal{N}_f|} = 459$.

$\lvert\mathcal{N}_f\rvert$	h	N	e	η	η/e	α_e	α_η
9	7.0711e-001	2	5.0321e-001	7.3306e+000	14.5679		
35	3.5355e-001	3	3.2739e-001	4.2363e+000	12.9398	0.6330	0.8075
135	1.7677e-001	6	1.6714e-001	2.1517e+000	12.8732	0.9960	1.0037
527	8.8385e-002	12	8.7291e-002	1.1113e+000	12.7308	0.9540	0.9703
2079	4.4193e-002	23	4.5373e-002	5.5936e-001	12.3282	0.9535	1.0004
8255	2.2097e-002	46	2.3185e-002	2.8227e-001	12.1744	0.9738	0.9920
32895	1.1048e-002	91	1.1687e-002	1.4125e-001	12.0866	0.9911	1.0015
131327	5.5240e-003	182	5.9178e-003	7.0856e-002	11.9732	0.9831	0.9967

Table 4.1: Numerical results for Example 4.1 with time discretization parameter $\theta = 1$, i.e. backward Euler, computed with a uniform strategy, $\tau = h$. Here, $|\mathcal{N}_f|$ denotes the number of free nodes, h denotes the diameter of the elements, N denotes the number of time steps, $e = |||u(1) - u_{h,\tau}(1)|||$, η is the error estimator, and α_e as well as α_η are the experimental convergence rates from (4.2)–(4.3).

Semi-uniform strategy: Correlation between time step size and spatial mesh size

For a fixed number of time steps $N \in \{9, \dots, 513\}$(according to $\tau \in \{1/8, \dots, 1/512\}$), we compute the error $e = |||u(1) - u_{h,\tau}(1)|||$ at final time $T = 1$ and the corresponding error estimator η for a sequence of uniformly refined meshes with $|\mathcal{N}_f| \in \{9, 35, 135, 527, 2079, 8255, 32895\}$. The computation are carried out with the backward Euler scheme ($\theta = 1$). In Figure 4.4, we plot the obtained values of error e and error estimator η over the square root $\sqrt{|\mathcal{N}_f|}$ of the number of free nodes, for each number of time steps N. The optimal convergence rate is indicated by the slope -1. In particular Figure 4.4 shows, that the curves for more time steps N converge better, in the sense that their slope is approximate -1 for computations with more free nodes.

Uniform strategy: Specifying the space-time correlation

For a uniform strategy with the backward Euler scheme, a convergence rate $e = |||u(1) - u_{h,\tau}(1)||| = \mathcal{O}(\tau + h)$ for the temporal and spatial meshsizes can be predicted from Theorem 2.6. This indicates a dependence $h = \tau^\sigma$ with $\sigma = 1$ for the optimal convergence of a uniform algorithm. To verify that the choice $\sigma = 1$ is asymptotically optimal, we compute the error e at final time $T = 1$ and the corresponding error estimator η with the uniform strategy: For prescribed time intervals $\tau = 1/8, \dots 1/64$, we uniformly refine the initial mesh as long as $h > \tau^\sigma$ for various $\sigma = 1/2, \dots, 3/2$. For $\sigma = 1/2, 3/4$ we compute the corresponding values even for $\tau = 1/128, 1/256$. We plot the values of the error and the error estimator obtained from this computation over the number of time steps in Figure 4.5. For $\sigma < 1$ we observe a suboptimal convergence rate. On the other hand, the computations with $\sigma \geq 1$ all show the optimal convergence rate $\mathcal{O}(\tau)$, visualized by the slope -1 in Figure 4.5. Thus, all choices $\sigma > 1$ lead to smaller time step size than necessary and therefore to a certain inefficiency of the algorithm. In this sense $\sigma = 1$ seems to be optimal.

Uniform strategy vs. adaptive strategy

Figure 4.6 shows the experimental results for the backward Euler method (θ=1) and visualizes the differences between uniform and adaptive strategy. For the uniform strategy, we choose $\tau = h$ and a uniform spatial mesh with mesh-size $h = \sqrt{0.5}$ for the initial step, c.f. Figure 4.2. For any further computation we refine the spatial mesh uniformly and we bisect the temporal mesh-size accordingly. See Table 4.2 for the numerical results of these computations. For final

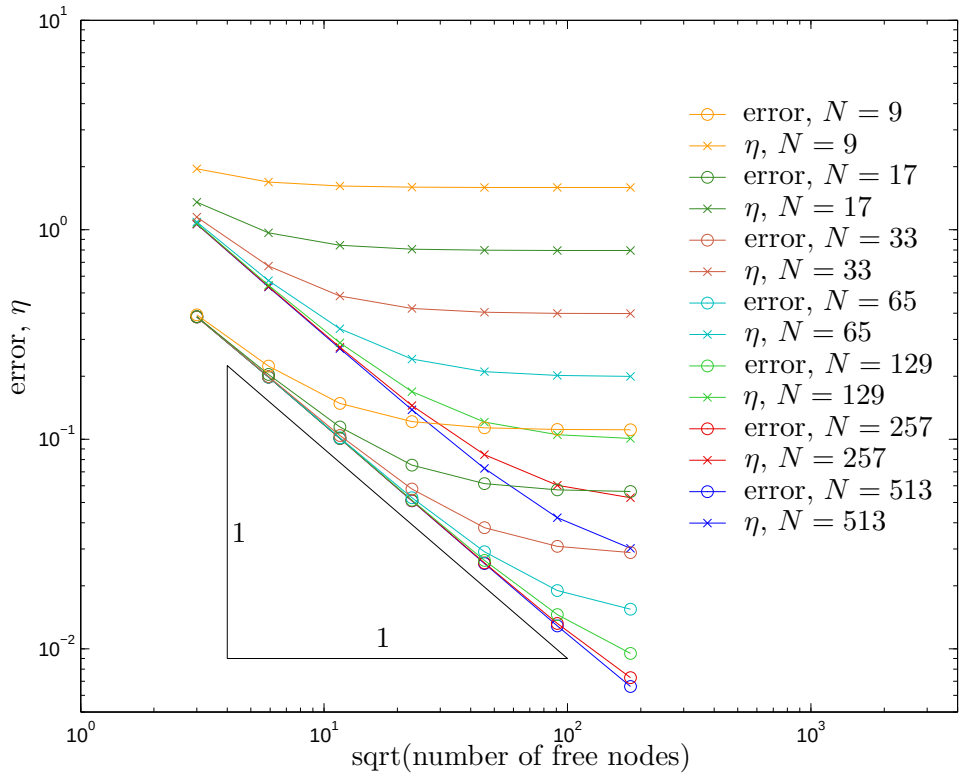


Figure 4.4: Error $\|u(1) - u_{h,\tau}(1)\|$ at final time $T = 1$ and corresponding error estimator obtained from a uniform strategy with $\theta = 1$ in Example 4.1. Each curve corresponds to computations with certain number of equidistant time steps $N \in \{9, \dots, 513\}$. The spatial meshes were obtained by uniform refinement of the initial mesh, c.f. Figure 4.2. For fixed number of time steps N, we plot the error and error estimator over the square root of the number of free nodes. The optimal convergence rate is indicated by the slope -1. We observe, that the curves with more time steps show this optimal convergence up to more free nodes. In this sense, there is a strong correlation between time step length τ and spatial mesh-size h.

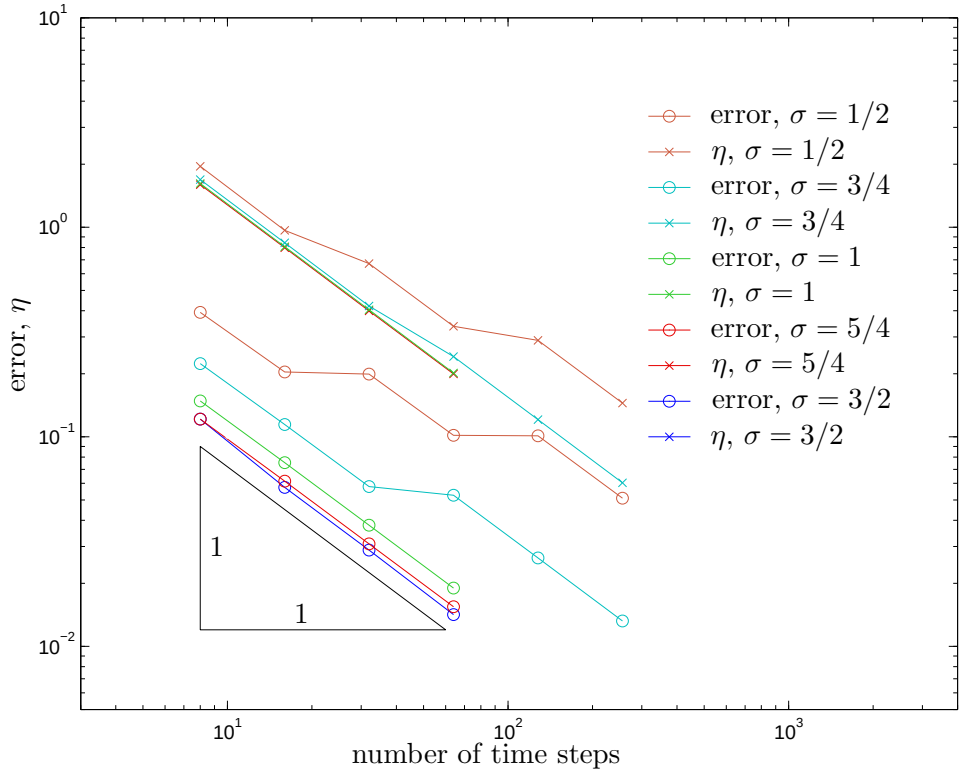


Figure 4.5: Error $\|u(1) - u_{h,\tau}(1)\|$ at final time $T = 1$ and corresponding error estimator obtained from a uniform strategy with $\theta = 1$ in Example 4.1. The meshes for this computation were obtained as follows: For certain time step sizes $\tau \in \{1/8, \dots, 1/256\}$ and an exponent $\sigma \in \{1/2, \dots, 3/2\}$, we uniformly red-refine all elements from the initial triangulation visualized in Figure 4.2 as long as $h > \tau^\sigma$. Each curve corresponds to a certain value of σ indicated in the legend. For $\sigma < 1$ we observe suboptimal convergence with respect to the number of time steps, whereas the curves with $\sigma \geq 1$ all show a convergence rate $\mathcal{O}(\tau)$, as indicated by the slope -1. Note that the computations were carried out up to $h \leq (1/64)^{3/2} \sim |\mathcal{N}_f|^{-1/2}$, which corresponds to a uniform refined mesh with 525.825 free nodes.

time $T = 1$, we then plot the obtained values of the error estimator η as well as of the error $e = \|\!|u(1) - u_{h,\tau}(1)|\!\|$ over the square root $\sqrt{|\mathcal{N}_f|}$ of the number of free nodes, in Figure 4.6. Note that the regularity of u predicts an order of convergence $e = \mathcal{O}(h + \tau) = \mathcal{O}(h)$ which corresponds to a slope -1 in the figure. This rate can be observed experimentally.
For the adaptive strategy, we use a sequence of tolerances $\varepsilon = 1/2, \ldots, 1/32$ and obtain the corresponding values of e and η as well a the mean number of free nodes $\overline{|\mathcal{N}_f|}$. We plot the error estimator and the error over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ thereof in Figure 4.6, since this quantity is related to $\sqrt{|\mathcal{N}_f|}$ in case of uniform computations. As for the uniform strategy, we observe optimal order of convergence in the sense that the visualization provides curves with the slope -1. However, we even see that the adaptive strategy is superior, since the absolute values of error and error estimator are smaller than for the uniform strategy. See Table 4.1 and 4.2 for the detailed numerical results of this experiment.
Next, we consider the adaptive strategy with time-discretization parameter $\theta = 1/2$, i.e. Crank-Nicholson scheme. A comparison between Table 4.3 and 4.2 indicates, that the Crank-Nicholson scheme uses less time steps, but at the cost of more free nodes per step.

Adaptive strategy: Correlation between time and number of free nodes

Since the curvature increases with respect to time by the order t^2, we expect the number of free nodes for a time step t_n to be proportional to t_n^4, where $n = 1, \ldots, N$. This can heuristically be seen as follows: The approximation theorem provides

$$\|\!|u(t_n) - u_{h,\tau}(t_n)|\!\| \sim \|h_{\mathcal{T}^{(n)}} D^2 u\| = \|h_{\mathcal{T}^{(n)}} \frac{\partial^2}{\partial x^2} u\| \sim \|h_{\mathcal{T}^{(n)}} t_n^2\|,$$

where $h_{\mathcal{T}^{(n)}} \in L^\infty(\Omega)$ is the local mesh-width function corresponding to $\mathcal{T}^{(n)}$, i.e. $h_{\mathcal{T}^{(n)}}|_K = h_K$ for all $K \in \mathcal{T}^{(n)}$, c.f. Braess [5, Chapter II, §7, Th. 7.3, p.86]. In order for the error $\|\!|u(t_n) - u_{h,\tau}(t_n)|\!\|$ at the time step t_n to stay dominated by the tolerance ε, $h_{\mathcal{T}^{(n)}}$ has to be of the order of t_n^{-2}. Thus,

$$t_n^2 \sim h_{\mathcal{T}^{(n)}}^{-1} \sim |\mathcal{N}_f^{(n)}|^{1/2}.$$

To visualize this correlation we plotted the time steps between 0 and final time 1 versus the number of free nodes in $\mathcal{T}^{(n)}$ used at the corresponding time step n for tolerance $\varepsilon = 1/4$ in Figure 4.7. Additionally, we plot the parabolic curve at^4 on the same time interval, where we fit the parameter a such that the parabola intersects 0 and the maximum number of free nodes at final time 1, that is $|\mathcal{N}_f^{(N)}|$. The same qualitative behavior is observed. Moreover, the figure indicates that there is no crucial coarsening in the solution process, since the number of free nodes seems to increase constantly with respect to time. As the computation shows, there is in fact no coarsening at all, since the time intervals keep decreasing, c.f. the space-time adaptive strategy from Section 3.7.

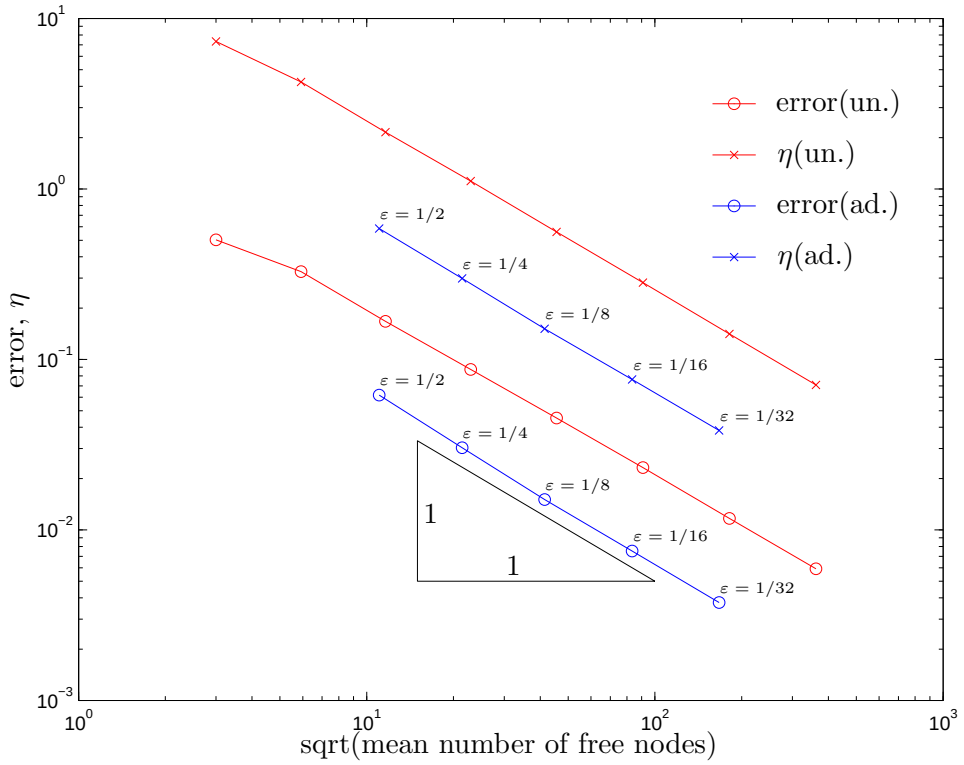


Figure 4.6: Error $\|u(1) - u_{h,\tau}(1)\|$ at final time $T = 1$ and corresponding error estimator η for the uniform strategy as well as the space-time adaptive strategy, for Example 4.1. All numerical computations were carried out with the time discretization parameter $\theta = 1$. The two curves from the uniform computation (red) are plotted over the square root $\sqrt{|\mathcal{N}_f|}$of the number of free nodes, where the time intervals are chosen to satisfy $\tau = h$. On the other hand, both curves from the adaptive strategy (blue), are plotted over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ of the mean number of free nodes, c.f. (4.1). We observe, that the uniform strategy leads to an experimental convergence rate of order $\mathcal{O}(|\mathcal{N}_f|^{-1/2}) = \mathcal{O}(h)$. For the adaptive strategy, the error and error estimator are computed for certain tolerances ε. Here, we observe the experimental convergence rate $\mathcal{O}(\overline{|\mathcal{N}_f|}^{-1/2})$. In particular, the adaptive strategy is superior to the uniform computation with regard to the absolute values of error and error estimator.

ε	$\overline{\lvert\mathcal{N}_f\rvert}$	N	e	η	η/e	α_e	α_η
1/2	122	26	6.1764e-002	5.8639e-001	9.5933		
1/4	459	51	3.0320e-002	2.9862e-001	9.9048	1.04	1.01
1/8	1715	101	1.5068e-002	1.5145e-001	10.0773	1.02	1.00
1/16	6954	202	7.5145e-003	7.6254e-002	10.1608	1.00	0.98
1/32	27872	404	3.7553e-003	3.8355e-002	10.2137	1.00	0.99

Table 4.2: Numerical results for Example 4.1 with the time discretization parameter $\theta = 1$, i.e. the backward Euler method, computed with the adaptive strategy. Here, $\overline{|\mathcal{N}_f|}$ denotes the mean number of free nodes, N denotes the number of time steps, $e = \|u(1) - u_{h,\tau}(1)\|$, η is the error estimator, and α_e as well as α_η are the experimental convergence rates from (4.4)–(4.5).

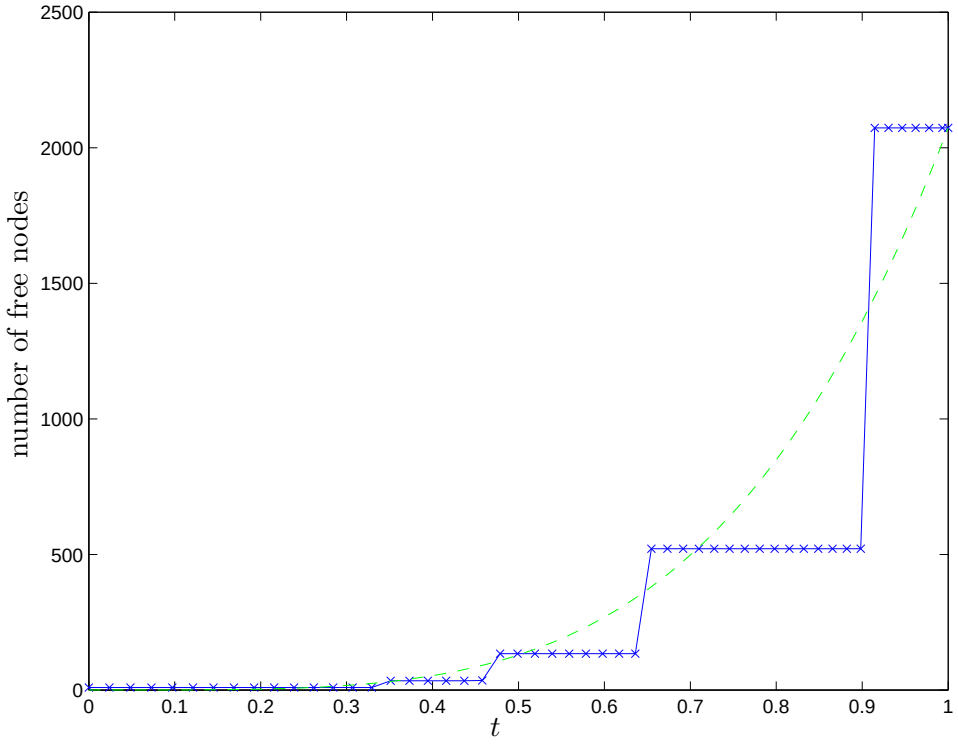


Figure 4.7: Number of free nodes $|\mathcal{N}_f^{(n)}|$ for each time step t_n in Example 4.1 between $t = 0$ and final time $T = 1$, where $\theta = 1$ and tolerance $\varepsilon = 1/4$ (blue). For comparison, we plot the parabolic curve $|\mathcal{N}_f^{(N)}|t^4$ (green) on the same time interval. We observe that both curves show the same qualitative behavior.

ε	$\overline{\lvert\mathcal{N}_f\rvert}$	N	e	η	$\eta/\lvert\!\lvert\!\lvert e\rvert\!\rvert\!\rvert$	α_e	α_η
1/2	221	21	7.5352e-002	6.4466e-001	8.5301		
1/4	644	40	3.9358e-002	3.3090e-001	8.4600	1.06	1.05
1/8	2632	80	1.9207e-002	1.6639e-001	8.6258	1.00	0.99
1/16	10385	160	9.9108e-003	8.3830e-002	8.5127	1.00	1.00
1/32	41381	321	4.8243e-003	4.2173e-002	8.7417	1.04	0.99

Table 4.3: Numerical results for Example 4.1 with the time discretization parameter $\theta = 1/2$, i.e. the Crank-Nicholson scheme, computed with the adaptive strategy. Here, $\overline{|\mathcal{N}_f|}$ denotes the mean number of free nodes, N denotes the number of time steps, $e = \|\!|\!| u(1) - u_{h,\tau}(1) |\!|\!|$, η is the error estimator, and α_e as well as α_η are the experimental convergence rates from (4.4)–(4.5).

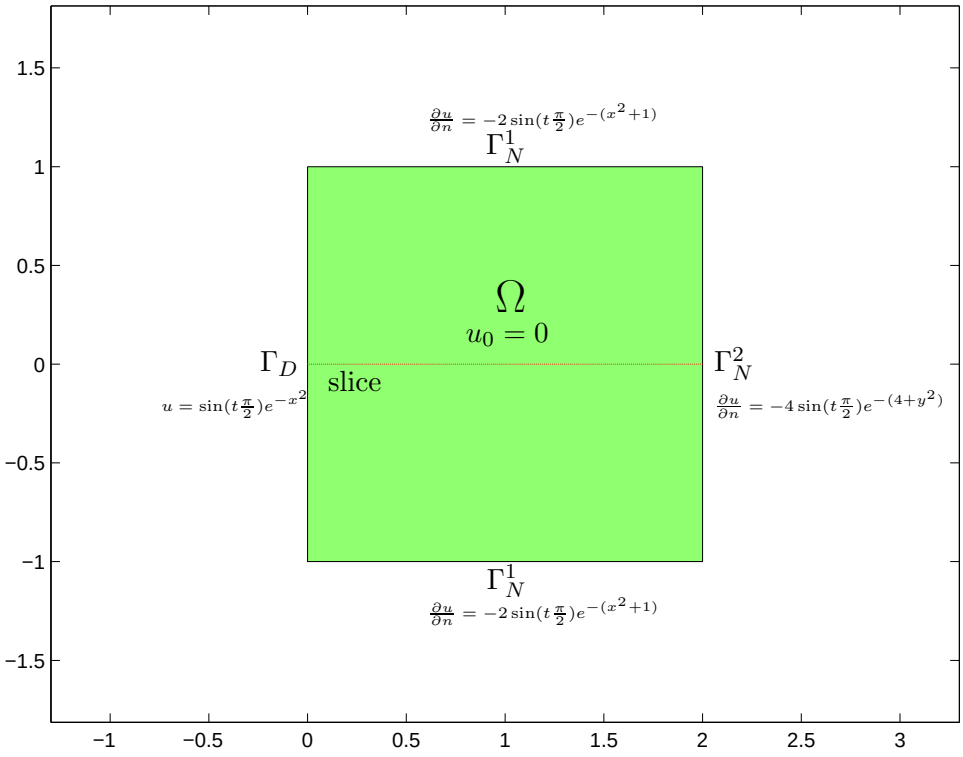


Figure 4.8: Geometry as well as initial and boundary data in Example 4.2.

4.2 A non-trivial Model Problem

As a further model problem, we consider the following linear parabolic equation with constant coefficients. Here, the prescribed solution reads $u(x, y, t) = u(\|\vec{x}\|, t) = \sin(t\pi/2)e^{-(x^2+y^2)}$ and initial condition u_0, boundary values and right hand side f follow thereof[2]. We carry out similar experiments as above, so we refer to Example 4.1 for a more detailed description.

$$
\begin{aligned}
\frac{\partial u}{\partial t} - \triangle u &= f \text{ in } \Omega \times (0,1], \\
u &= \sin\big(t\frac{\pi}{2}\big)e^{-x^2} \text{ on } \Gamma_D \times (0,1], \\
\vec{n} \cdot \nabla u &= \begin{cases} -2\sin(t\frac{\pi}{2})e^{-(x^2+1)} & \text{on } \Gamma_N^1 \times (0,1], \\ -4\sin(t\frac{\pi}{2})e^{-(4+y^2)} & \text{on } \Gamma_N^2 \times (0,1], \end{cases} \\
u_0 &= 0 \text{ in } \Omega,
\end{aligned}
\tag{4.7}
$$

where $\Omega := [0,-1] \times [2,1]$, $\Gamma_D := \{0\} \times (-1,1)$, $\Gamma_N^1 := (0,1) \times \{-1\} \cup (0,1) \times \{1\}$, $\Gamma_N^2 := \{2\} \times (-1,1)$.

The setting of the problem is visualized in Figure 4.8. In Figure 4.9, we plot the initial mesh used in all uniform as well as in all adaptive computations. We choose the time discretization parameter $\theta = 1$, i.e. backward Euler method. In 4.10, we plot the values of $u_{h,\tau}$ at final time $T = 1$ obtained by an adaptive computation with tolerance $\varepsilon = 1/8$. Since we use P^1-finite elements, the discrete solution $u_{h,\tau}$ can not be radially symmetric. However, we observe in

[2]The volume force reads $f = -\pi/2\cos(t\pi/2)\exp[-(x^2+y^2)] - 4\sin(t\pi/2)\exp[-(x^2+y^2)](x^2+y^2-1)$.

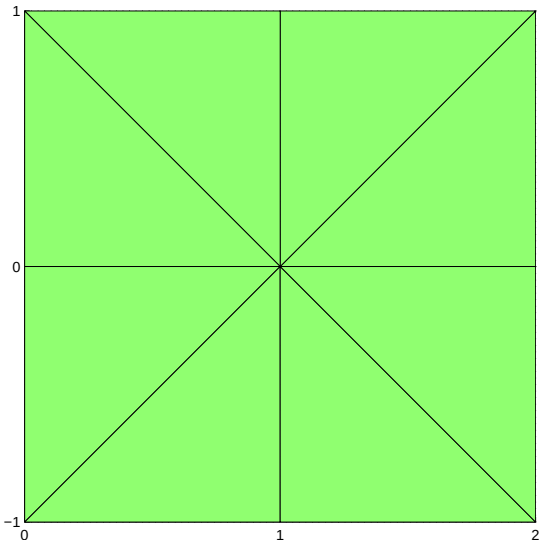


Figure 4.9: Initial triangulation in Example 4.2 with 8 elements and $|\mathcal{N}_f| = 6$ free nodes.

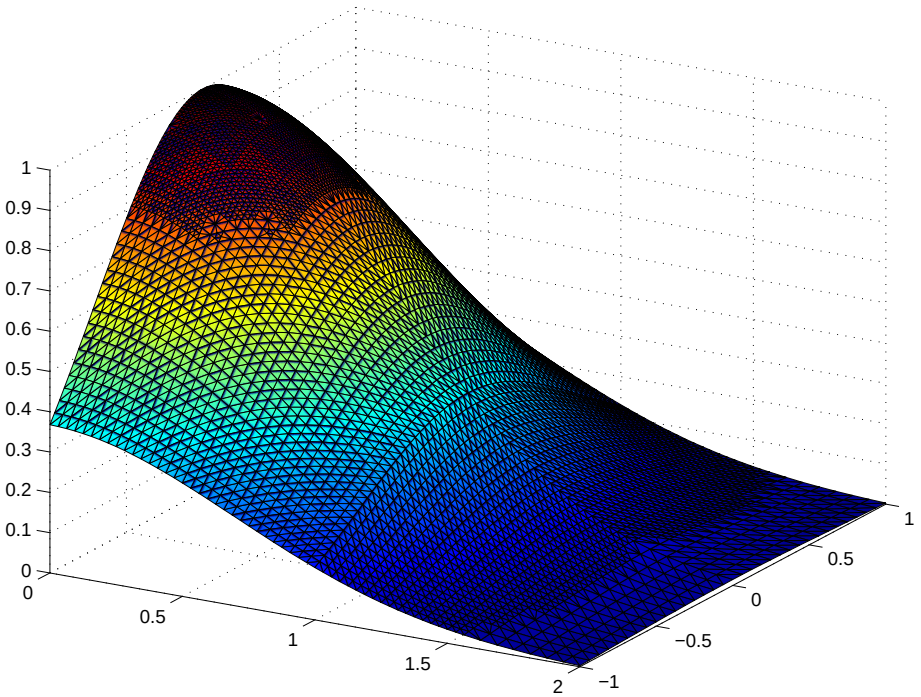


Figure 4.10: Plot of the discrete solution in Example 4.2 at final time $T = 1$ for an adaptive computation with time-discretization parameter $\theta = 1$ and tolerance $\varepsilon = 1/8$. The computation leads to a final mesh $\mathcal{T}^{(39)}$ with 9510 elements and $|\mathcal{N}_f| = 4874$ free nodes. Moreover, the adaptive algorithm needed $N = 39$ time steps and an average of $\overline{|\mathcal{N}_f|} = 1950$ free nodes.

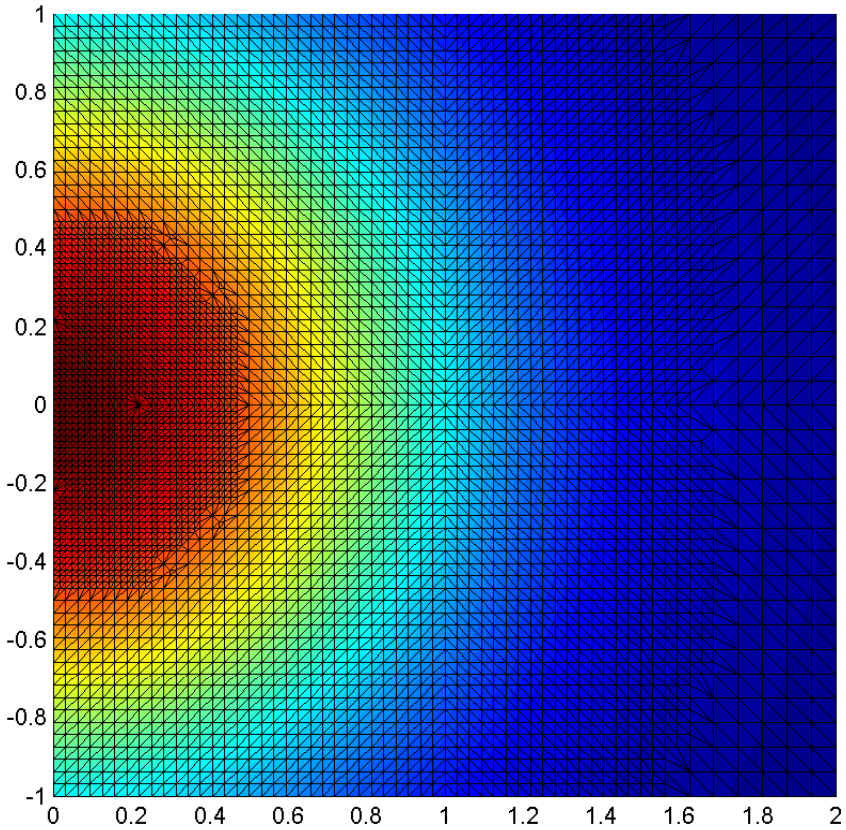


Figure 4.11: Projection of the discrete solution in Example 4.2 at final time $T = 1$ for an adaptive computation with time-discretization parameter $\theta = 1$ and tolerance $\varepsilon = 1/8$ onto the $x - y$ plane. The computation leads to a final mesh with 9510 elements and $|\mathcal{N}_f| = 4874$ free nodes. A surface plot of $u_{h,\tau}(1)$ is shown in Figure 4.10.

Figure 4.11 that symmetry with respect to the x-axis is achieved. Here, we plot the projection of $u_{h,\tau}(1)$ onto the $x - y$ plane.

Uniform strategy vs. adaptive strategy

Figure 4.12 shows the experimental results and visualizes the differences between uniform and adaptive strategy. For the uniform strategy, we choose $\tau = h$ and the uniform spatial mesh with mesh-size $h = \sqrt{2}$ for the initial step, which is shown in Figure 4.9. Analogously to Section 4.1, we refine the spatial mesh uniformly and we bisect the temporal mesh-size accordingly, for any further uniform computation.
For the adaptive strategy, we use a sequence of tolerances $\varepsilon = 1/2, \dots, 1/32$ and obtain the corresponding values of e and η as well a the mean number of free nodes $\overline{|\mathcal{N}_f|}$. The detailed outcome of these computations are provided in Table 4.4 and 4.5. In Figure 4.6, we then plot, the obtained values of the error estimator η for final time $T = 1$ as well as of the error $e = |||u(1) - u_{h,\tau}(1)|||$ over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ of the mean number of free nodes. Again, the regularity of u predicts an order of convergence $e = \mathcal{O}(h + \tau) = \mathcal{O}(h)$ which corresponds to a slope -1 in the figure. This rate is in fact observed in this experiment.

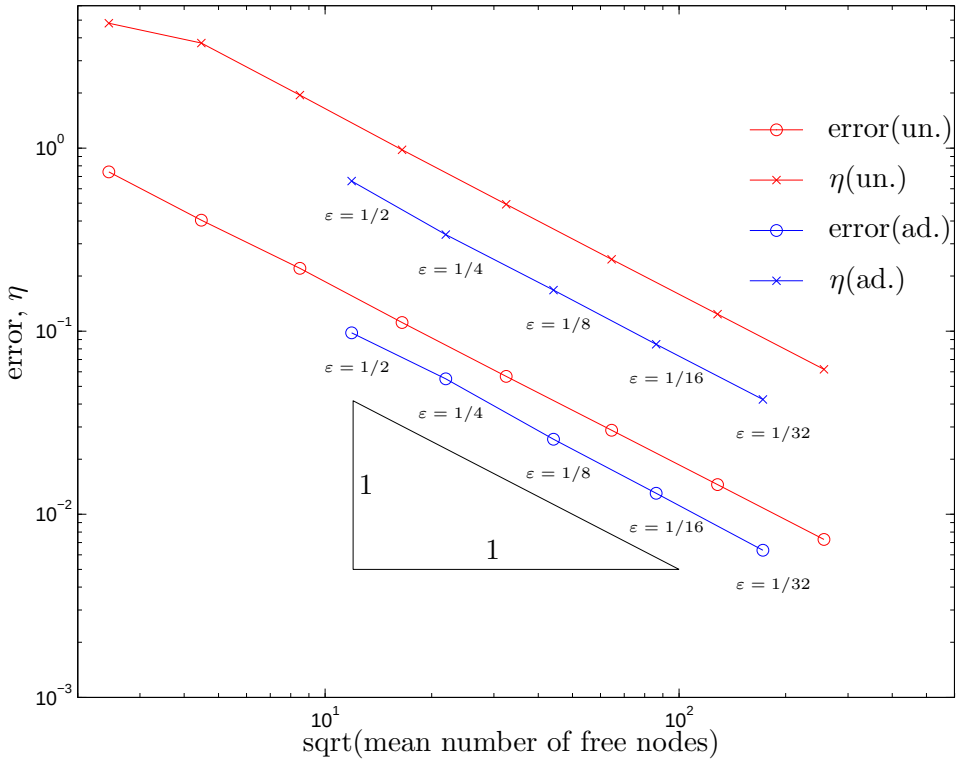


Figure 4.12: Error $\|\!|u(1) - u_{h,\tau}(1)|\!\|$ at final time $T = 1$ and corresponding error estimator η for the uniform strategy with $h = \tau$ as well as the space-time adaptive strategy, in Example 4.2. All numerical computations were carried out with the time discretization parameter $\theta = 1$. The two curves from the uniform computation (red) are plotted over the square root $\sqrt{|\mathcal{N}_f|}$ of the number of free nodes, where the time intervals are chosen to satisfy $\tau = h$. On the other hand, both curves from the adaptive strategy (blue), are plotted over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ of the mean number of free nodes, c.f. (4.1). We observe, that the uniform strategy leads to an experimental convergence rate of order $\mathcal{O}(|\mathcal{N}_f|^{-1/2}) = \mathcal{O}(h)$. For the adaptive strategy, the error and error estimator are computed for certain tolerances ε. Here, we observe the experimental convergence rate $\mathcal{O}(\overline{|\mathcal{N}_f|}^{-1/2})$. Moreover, the adaptive strategy is superior to the uniform computation with regard to the absolute values of error and error estimator.

ε	$\overline{\lvert\mathcal{N}_f\rvert}$	N	$\|\!\|e\|\!\|$	η	$\eta/\|\!\|e\|\!\|$	α_e	α_η
1/2	141	10	9.7940e-002	6.6100e-001	6.5961		
1/4	481	20	5.4948e-002	3.3674e-001	6.2187	1.02	1.03
1/8	1950	39	2.5674e-002	1.6763e-001	6.5321	1.05	1.01
1/16	7418	77	1.3016e-002	8.4766e-002	6.5414	1.02	1.01
1/32	29740	154	6.3652e-003	4.2413e-002	6.6634	1.03	1.00

Table 4.4: Numerical results for Example 4.2 with time discretization parameter $\theta = 1$, i.e. the backward Euler method, in the adaptive strategy. Here, $\overline{|\mathcal{N}_f|}$ denotes the mean number of free nodes, N denotes the number of time steps, $e = \|\!|u(1) - u_{h,\tau}(1)|\!\|$, η is the error estimator, and α_e as well as α_η are the experimental convergence rates from (4.4)–(4.5).

$\lvert\mathcal{N}_f\rvert$	h	N	e	η	η/e	α_e	α_η
6	1.4142e+000	1	7.4081e-001	4.8125e+000	6.4963		
20	7.0711e-001	2	4.0380e-001	3.7472e+000	9.2798	1.0080	0.4157
72	3.5355e-001	3	2.2029e-001	1.9481e+000	8.8433	0.9461	1.0213
272	1.7678e-001	6	1.1146e-001	9.7875e-001	8.7808	1.0251	1.0358
1056	8.8388e-002	12	5.6657e-002	4.9333e-001	8.7073	0.9977	1.0101
4160	4.4194e-002	23	2.8761e-002	2.4708e-001	8.5908	0.9890	1.0087
16512	2.2097e-002	46	1.4510e-002	1.2386e-001	8.5363	0.9926	1.0018
65792	1.1049e-002	91	7.2797e-003	6.1934e-002	8.5078	0.9979	1.0028

Table 4.5: Numerical results for a uniform strategy with $\tau = h$ in Example 4.2 with time discretization parameter $\theta = 1$, i.e. backward Euler method. Here, $|\mathcal{N}_f|$ denotes the number of free nodes, h denotes the diameter of the elements, N denotes the number of time steps, $e = \|u(1) - u_{h,\tau}(1)\|$, η is the error estimator, and α_e as well as α_η are the experimental convergence rates from (4.2)–(4.3).

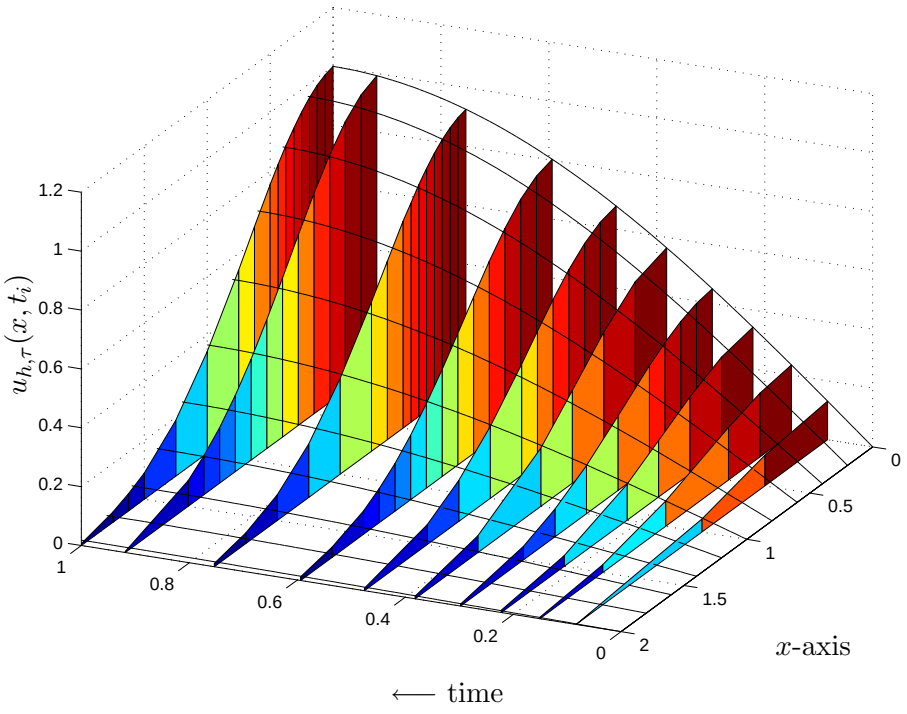


Figure 4.13: Visualization of the discrete solution $u_{h,\tau}$ of Example 4.2, along the slice $\mathrm{conv}\{(0,0),(2,0)\} \subseteq \overline{\Omega}$ in the $y-z$ plane for different time steps $t_i \in (0,1]$ indicated on the y-axis. Here $x = 0$, $x = 2$ correspond to $(0,0)$ and $(2,0)$, respectively, c.f. Figure (4.10). The slice is also indicated in Figure 4.8. The black curves show the analytic solution $\sin(t\pi/2)e^{-x^2}$, plotted over time and for various equidistant x values.

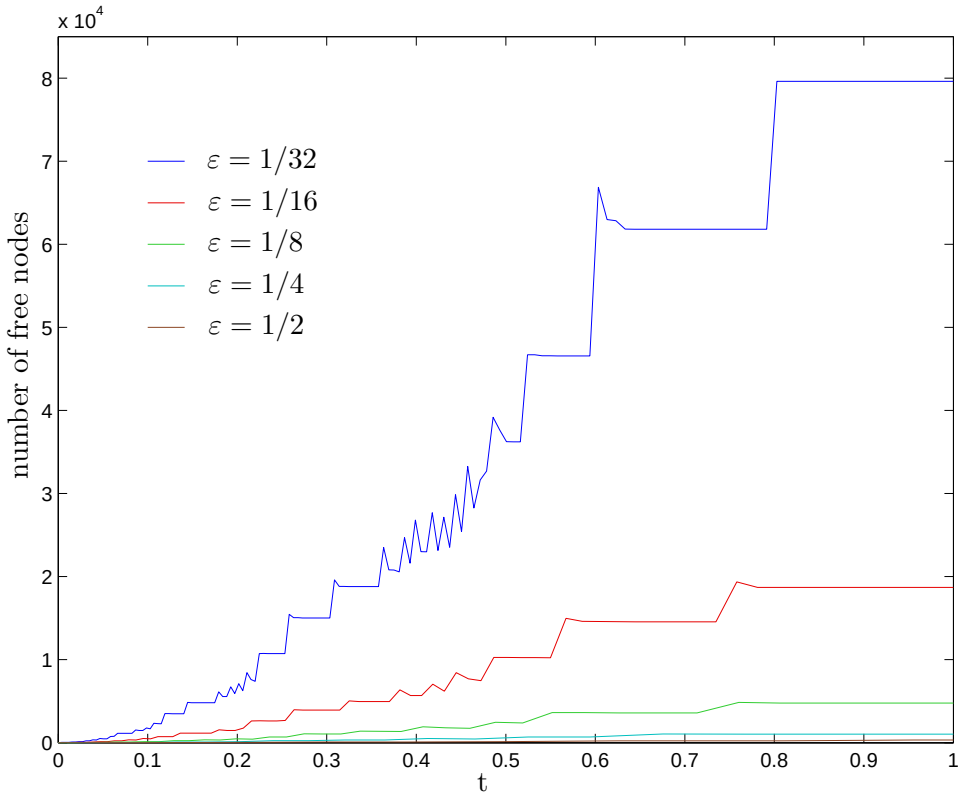


Figure 4.14: Number of free nodes $|\mathcal{N}_f^{(n)}|$ for each time step t_n in Example 4.2 between $t = 0$ and final time $T = 1$, used by the adaptive algorithm with time discretization parameter $\theta = 1$ and tolerances $\varepsilon \in \{1/2, \dots, 1/32\}$. For visualization the discrete values of $|\mathcal{N}_f^{(n)}|$ are linearly interpolated.

Adaptive strategy: Visualizing the dynamics

For the dynamics of the problem we first focus on a specific slice of the domain, indicated as "slice" in Figure 4.8. We plot the discrete solution $u_{h,\tau}$ on $\operatorname{conv}\{(0,0),(2,0)\} \subseteq \overline{\Omega}$ for the time steps computed by the adaptive algorithm with $\varepsilon = 1/2$ in Figure 4.13. The analytic solution on this slice reads $\sin(t\pi/2)e^{-x^2}$ and is plotted by black curves into Figure 4.13. We observe the same qualitative behavior. For all drawn slices in Figure 4.13, each colored part corresponds to one triangle. We observe an increase of elements with respect to space for smaller x and with respect to time spatially global.

In Figure 4.14 we observe, that the number of free nodes decreases temporally in the solution process, i.e. the triangulations have been coarsened. Here, we plot the time steps t_n versus the local number of free nodes $|\mathcal{N}_f^{(n)}|$ which are needed by the adaptive algorithm for tolerances $\varepsilon = 1/2, \dots, 1/32$. For the ease of visualization the discrete values of $|\mathcal{N}_f^{(n)}|$ are linearly interpolated therein. Finally, in Figure 4.15, we plot the time steps t_{n-1} versus the local time step size τ_n used by the adaptive algorithm for tolerances $\varepsilon = 1/2, \dots, 1/32$. The discrete values of τ_n are again linearly interpolated and we do not plot the final time step size τ_N for the ease of presentation. In the curves for all tolerances we observe, that τ_n attains a maximum at $t \approx 0.7$.

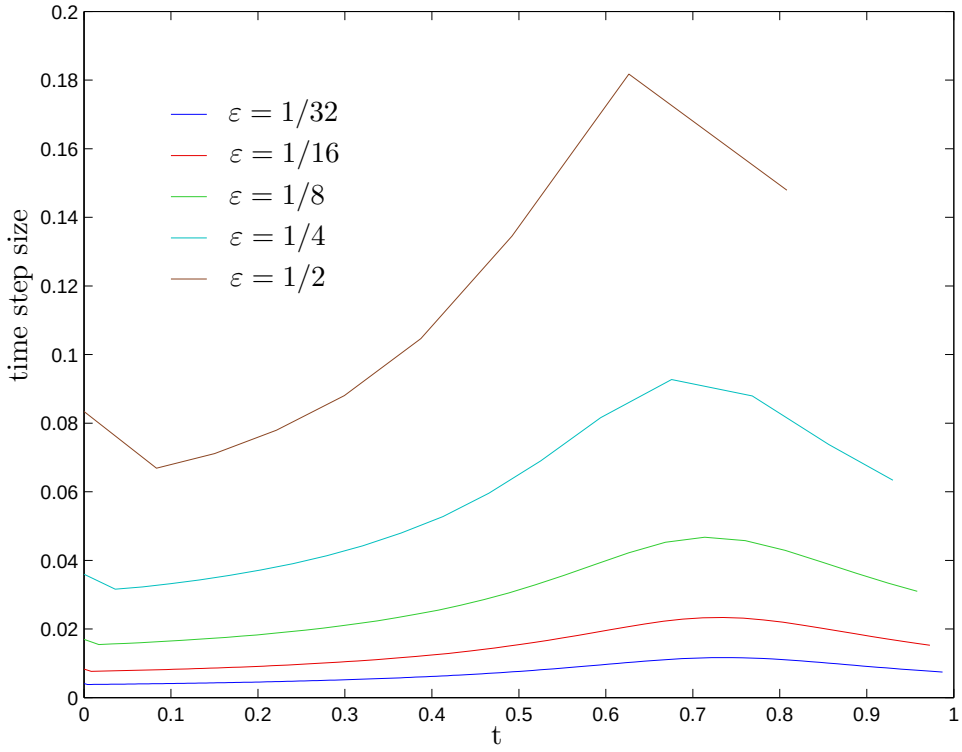


Figure 4.15: Time step sizes τ_n for each time step t_{n-1} in Example 4.2 between $t = 0$ and final time $T = 1$, used by the adaptive algorithm with time discretization parameter $\theta = 1$ and tolerances $\varepsilon \in \{1/2, \ldots, 1/32\}$. For visualization the discrete values of τ_n are linearly interpolated and we do not plot the final time step size τ_N.

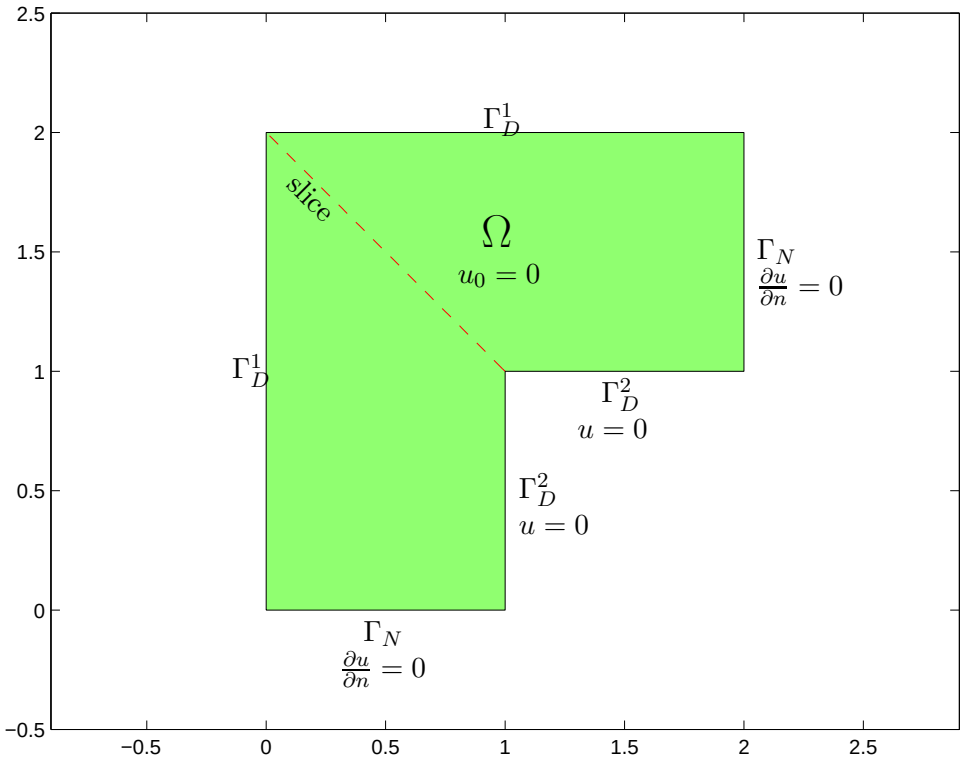


Figure 4.16: Geometry, as well as initial and boundary data in Example 4.3.

4.3 Heat Equation

We now approach a problem, where the analytic solution is unknown. Conducting similar experiments as above, we refer to Example 4.1 for a more detailed description. We consider the heat equation on the L-shaped domain shown in Figure 4.16:

$$
\begin{aligned}
&\frac{\partial u}{\partial t} - \triangle u = 0 \text{ in } \Omega \times (0,1], \\
&\qquad u = \begin{cases} t/0.01 & \text{on } \Gamma_D^1 \times (0, 0.01], \\ 1 & \text{on } \Gamma_D^1 \times [0.01, 1], \\ 0 & \text{on } \Gamma_D^2 \times (0,1], \end{cases} \\
&\vec{n} \cdot \nabla u = 0 \text{ on } \Gamma_N \times (0,1] \\
&\qquad u_0 = 0 \text{ in } \Omega,
\end{aligned}
\tag{4.8}
$$

where $\Omega := \{[0,0] \times [2,2]\} \setminus \{[1,1] \times [2,1]\}$, $\Gamma_D^1 := (0,0) \times \{0\} \cup (2,0) \times \{2\}$, $\Gamma_D^2 := (1,0) \times \{1\} \cup (1,1) \times \{1\}$, $\Gamma_N := (0,0) \times \{0\} \cup (2,2) \times \{2\}$.

Note that the non-trivial time-dependent Dirichlet boundary conditions correspond to a heat flow that is switched on instantly. The general setting of this problem is shown in Figure 4.16. The computations in this experiment are carried out with the time discretization parameter $\theta = 1$ (backward Euler). As initial triangulation for all uniform and for all adaptive computations, we use the mesh from Figure 4.17 with 24 elements and $|\mathcal{N}_f| = 7$ free nodes. In Figure 4.18 and 4.19, we plot the values of $u_{h,\tau}$ at final time $T = 1$ obtained by a adaptive computation with tolerance $\varepsilon = 1$. In Figure 4.19 we observe, that the adaptive algorithm leads to a final

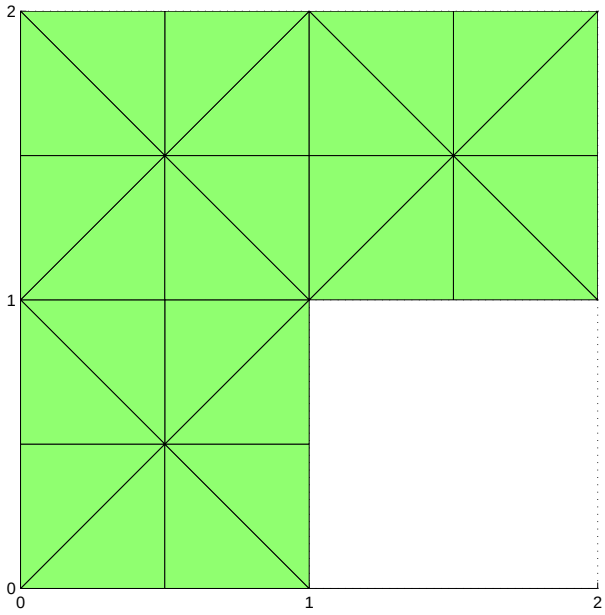


Figure 4.17: Initial triangulation in Example 4.3 with 24 elements and $|\mathcal{N}_f| = 7$ free nodes.

mesh $\mathcal{T}^{(12)}$ that is, in particular, symmetric with respect to the slice $\text{conv}\{(1,1),(0,2)\}$, which is indicated in Figure 4.16.

Uniform strategy vs. adaptive strategy

For the uniform strategy, we choose $\tau = h$ and a uniform spatial mesh with mesh-size $h = \sqrt{0.5}$ for the initial step, c.f. Figure 4.17. For the adaptive strategy, we use a sequence of tolerances $\varepsilon = 2, \dots, 1/8$ and obtain the corresponding values of η as well a the mean number of free nodes $\overline{|\mathcal{N}_f|}$. Figure 4.20 shows the experimental results and visualizes the differences between uniform and adaptive strategy. The numerical results of these computations can be found in Table 4.6 and 4.7. For final time $T = 1$, we then plot the obtained values of the error estimator η over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ of the mean number of free nodes, in Figure 4.20. The optimal convergence rate is indicated by the slope -1 therein.
For the uniform strategy, we observe in Figure 4.20 and Table 4.7 that the experimental convergence rate α_η constantly decreases starting from 0.39 to 0.25 until 0.84 for the final computation. With a time step size of $\tau = h = 0.0055$, this is the first computation where the initial time interval $(0, 0.01)$, when heat is switched on, is resolved. We conclude, that for the quality of the approximation obtained from adaptive computations, it seems vital to start with an initial first guess time step size $\widehat{\tau}_1 < 0.01$.
For all adaptive computations, we choose an initial first guess time step size $\widehat{\tau}_1 = 0.001$ in contrast to the original adaptive strategy, where $\widehat{\tau}_1 = \min\{T, 2, \varepsilon\}$. In Figure 4.20 and Table 4.6, we observe that the adaptive strategy is superior to the uniform strategy with respect to the experimental convergence rate. Moreover, except the value of η for $\varepsilon = 2$, the adaptive algorithm is superior to uniform computations with similar number of free nodes with respect to the absolute values of η.

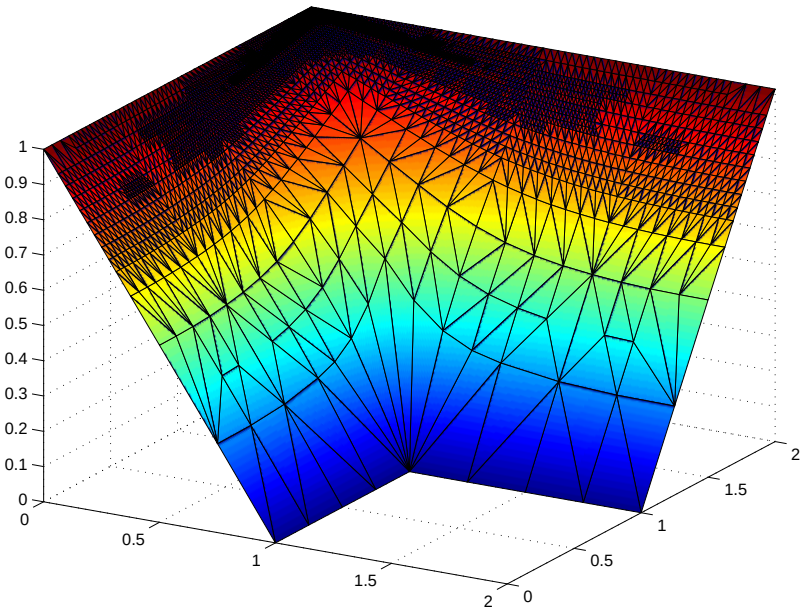


Figure 4.18: Discrete solution in Example 4.3 at final time $T = 1$ for an adaptive computation with time-discretization parameter $\theta = 1$ and tolerance $\varepsilon = 1$. The computation leads to a final mesh $\mathcal{T}^{(12)}$ with 7546 elements and 3879 free nodes. Moreover, the algorithm used $N = 12$ time steps with an average of $\overline{|\mathcal{N}_f|} = 5040$ free nodes.

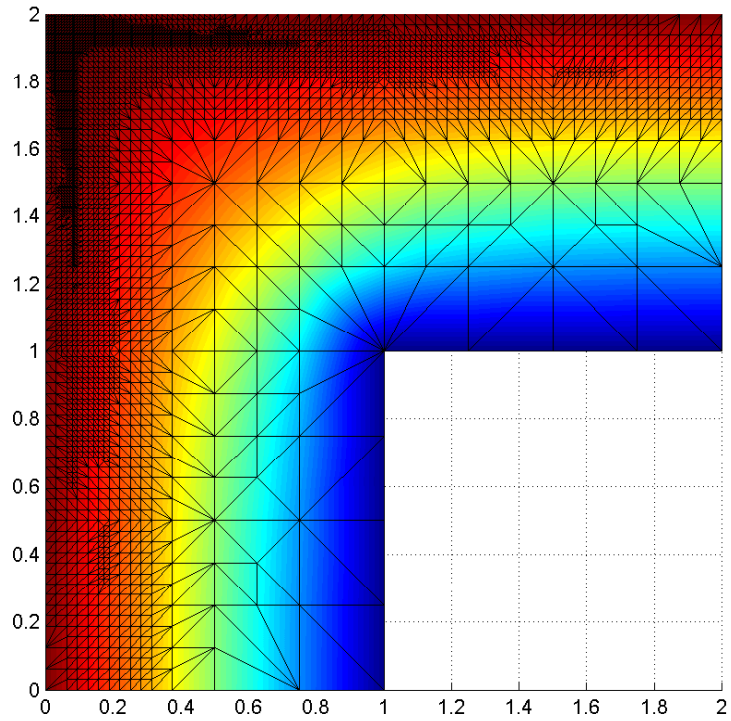


Figure 4.19: Projection of the discrete solution in Example 4.3 at final time $T = 1$ for an adaptive computation with time-discretization parameter $\theta = 1$ and tolerance $\varepsilon = 1$ onto the $x - y$ plane. The computation leads to a final mesh with 7546 elements and 3879 free nodes. A corresponding surface plot can be found in Figure 4.18.

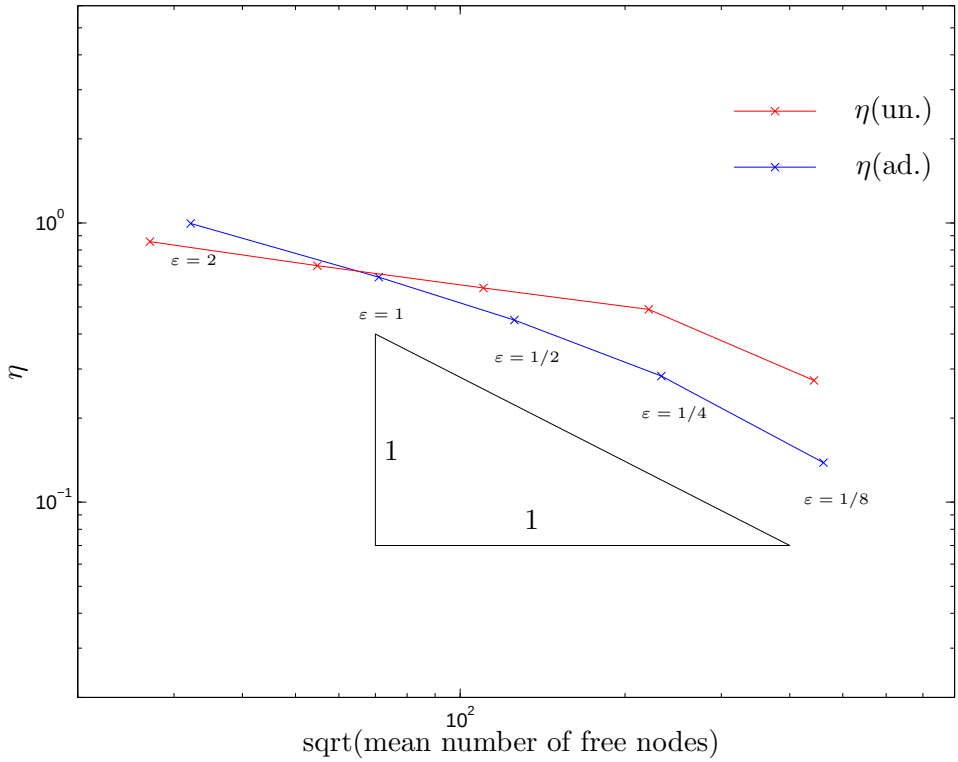


Figure 4.20: Error estimator η at final time $T = 1$ for the uniform strategy as well as for the adaptive strategy, for Example 4.3. All numerical computations are performed with time discretization parameter $\theta = 1$. The curve from the uniform computation (red) is plotted over the square root $\sqrt{|\mathcal{N}_f|}$ of the number of free nodes, where the time intervals are chosen to satisfy $\tau = h$. The curve from the adaptive strategy (blue), is plotted over the square root $\sqrt{\overline{|\mathcal{N}_f|}}$ of the mean number of free nodes, c.f. (4.1). We observe, that the uniform strategy leads to a suboptimal experimental convergence rate. For the adaptive strategy, the error estimator is computed for certain tolerances $\varepsilon = 2, \dots, 1/8$. Starting with a slightly suboptimal experimental convergence rate, the adaptive strategy tends towards the optimal rate $\mathcal{O}(\overline{|\mathcal{N}_f|}^{-1/2})$ for smaller ε. Except for the computation with $\varepsilon = 2$, the adaptive strategy is superior to the uniform computation with regard to the absolute values of the error estimator.

ε	$\overline{\lvert\mathcal{N}_f\rvert}$	N	η	α_η
2	1035	7	9.9615e-001	
1	5040	12	6.3973e-001	0.69
1/2	15767	23	4.4864e-001	0.67
1/4	54312	44	2.8290e-001	0.75
1/8	212584	90	1.3859e-001	1.05

Table 4.6: Numerical results for Example 4.3 with the time discretization parameter $\theta = 1$, i.e. the backward Euler method, computed with the adaptive strategy. Here, $\overline{|\mathcal{N}_f|}$ denotes the mean number of free nodes, N denotes the number of time steps, η is the error estimator, and α_η is the experimental convergence rate from (4.5).

$\overline{\lvert\mathcal{N}_f\rvert}$	h	N	η	α_η
7	7.0711e-001	2	2.0270e+000	
39	3.5355e-001	3	1.4483e+000	0.39
175	1.7678e-001	6	1.0837e+000	0.39
735	8.8388e-002	12	8.5735e-001	0.33
3007	4.4194e-002	23	7.0236e-001	0.28
12159	2.2097e-002	46	5.8470e-001	0.26
48895	1.1049e-002	91	4.9035e-001	0.25
196095	5.5243e-003	182	2.7293e-001	0.84

Table 4.7: Numerical results for Example 4.3 with the time discretization parameter $\theta = 1$, i.e. the backward Euler method, computed with a uniform strategy, $h = \tau$. Here, $\overline{|\mathcal{N}_f|}$ denotes the number of free nodes, h the diameter of the elements, N denotes the number of time steps, η is the error estimator, and α_η is the experimental convergence rate from (4.5).

Adaptive strategy: Suboptimal experimental convergence rate

We stress that the experimental convergence rate for $\varepsilon = 2, \dots, 1/4$, c.f. Figure 4.20 and Table 4.6, are suboptimal. Figure 4.18 and 4.19 for an adaptive computation with $\varepsilon = 1$ show, that the refinement level near the Dirichlet boundary Γ_D^1 is still rather high, though the system has clearly reached its steady state. A possible explanation is, that there were too few time steps for coarsening towards the end of the time interval. Because of the restriction in coarsening, i.e. only one coarsening step per time step, it becomes obvious that the algorithm has a certain inefficiency in problems with quickly smoothing solution. On the other hand, we observe in Figure 4.20 and Table 4.6 that the experimental convergence rate tends towards the optimal value 1 for smaller ε. Thus, it seems that smaller tolerances force the algorithm to use enough time steps for sufficient coarsening processes during the smoothing of the discrete solution $u_{h,\tau}$.

Adaptive strategy: Visualizing the dynamics

In Figure 4.21, we plot the time steps t_n versus the local number of free nodes $|\mathcal{N}_f^{(n)}|$ which are needed by the adaptive algorithm for tolerances $\varepsilon = 2, \dots, 1/8$. For the ease of visualization the discrete values of $|\mathcal{N}_f^{(n)}|$ are linearly interpolated therein. The curves for all tolerances show peaks at the time interval $[0, 0.01]$, when the heat is switched on. Towards the end of the time interval $(0, T]$ it seems as if there are too few time steps for further coarsening processes. Furthermore, in Figure 4.22, we plot the time steps t_{n-1} versus the local time step size τ_n used by the adaptive algorithm for tolerances $\varepsilon = 2, \dots, 1/8$. The discrete values of τ_n are again linearly interpolated and we do not plot the final time step size τ_N for the ease of presentation. We observe, that the majority of time steps are used in the initial interval $[0, 0.1]$, which includes the interval where the heat is switched on. Finally, we focus on the time evolution of the slice $\mathrm{conv}\{(1,1),(0,2)\}$, which is also indicated in Figure 4.16. We plot the values of the discrete solution $u_{h,\tau}$ along this slice for the time steps used by an adaptive computation with tolerance $\varepsilon = 1$ in Figure 4.23. Each colored area in this visualization corresponds to an element which lies on the slice. We observe that the density of triangles decreases with respect to time. However, as discussed above, in the steady state at final time $T = 1$ the elements still lie very dense near the Dirichlet boundary Γ_D^1. This indicates the inefficiency due to the lack of coarsening processes towards the end of the time interval $(0, 1]$.
Analytically, we expect the solution to reach a steady state within the time interval $[0, 1]$. Figure 4.23 and the other experiments indicate that the discrete solutions $u_{h,\tau}$ attains its steady state

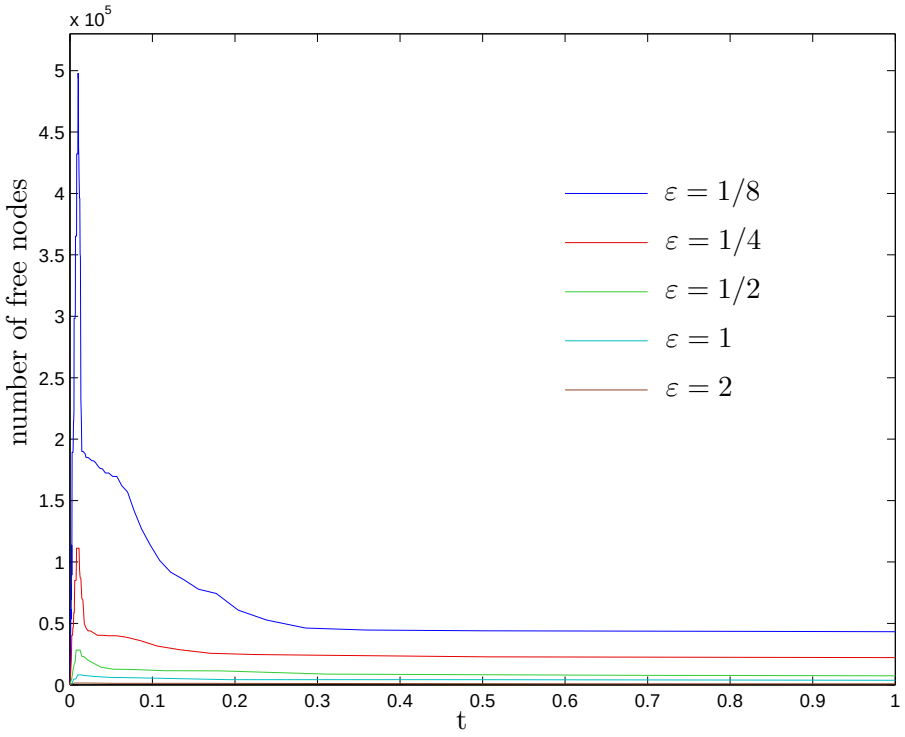


Figure 4.21: Number of free nodes $|\mathcal{N}_f^{(n)}|$ for each time step t_n in Example 4.3 between $t = 0$ and final time $T = 1$, used by the adaptive algorithm with time discretization parameter $\theta = 1$ and tolerances $\varepsilon \in \{2, \dots, 1/8\}$. For visualization the discrete values of $|\mathcal{N}_f^{(n)}|$ are linearly interpolated.

at $t \approx 0.5$. In additional numerical experiments, we observe that changing the final time $T = 10$ or $T = 100$ does neither change the discrete solution visibly nor does it increase the number of time steps. In this example, the adaptive algorithm from Section 3.7 thus shows an optimal time step size control, in the sense that additional computations are avoided, when the discrete solution has already reached equilibrium.

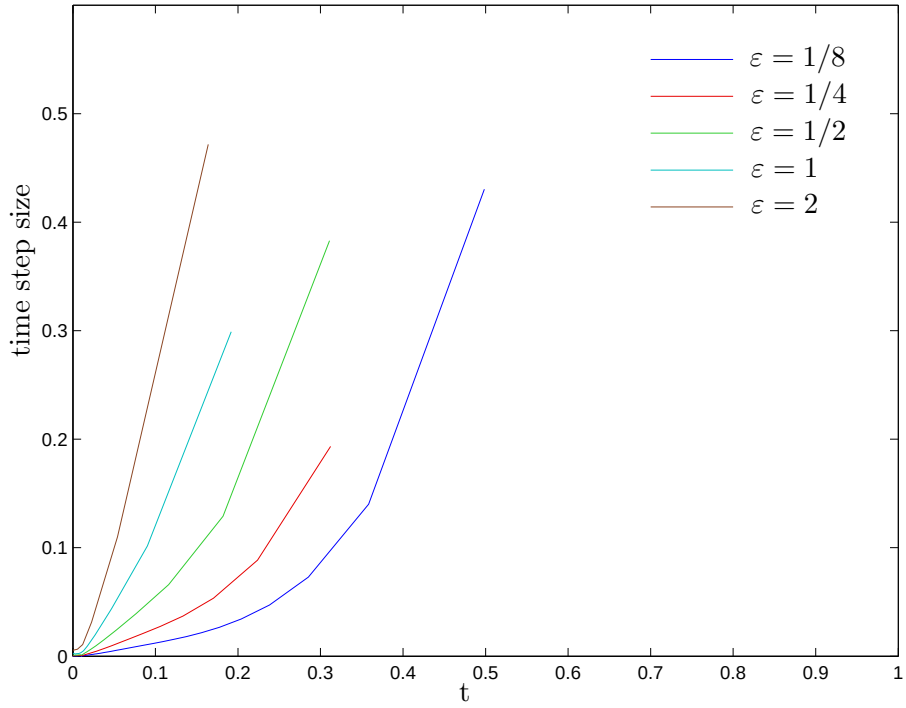


Figure 4.22: Time step sizes τ_n for each time step t_{n-1} in Example 4.3 between $t = 0$ and final time $T = 1$, used by the adaptive algorithm with time discretization parameter $\theta = 1$ and tolerances $\varepsilon \in \{2, \dots, 1/8\}$. For visualization the discrete values of τ_n are linearly interpolated and we do not plot the final time step size τ_N.

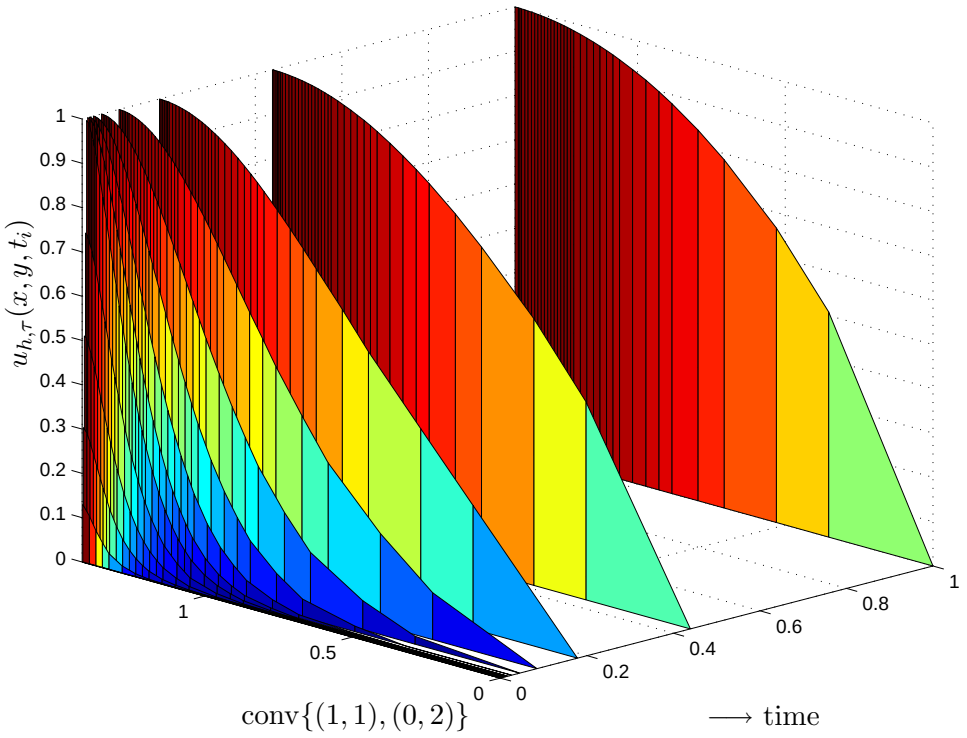


Figure 4.23: Visualization of the discrete solution $u_{h,\tau}$ in Example 4.3 along the slice $\text{conv}\{(1,1),(0,2)\} \subseteq \overline{\Omega}$ for different time steps $t_i \in (0, 1]$ indicated on the x-axis, c.f. Figure 4.16, where the slice is explicitly illustrated. Here $y = 0$, $y = \sqrt{2}$ correspond to $(1, 1)$ and $(0, 2)$, respectively, in Figure 4.18.

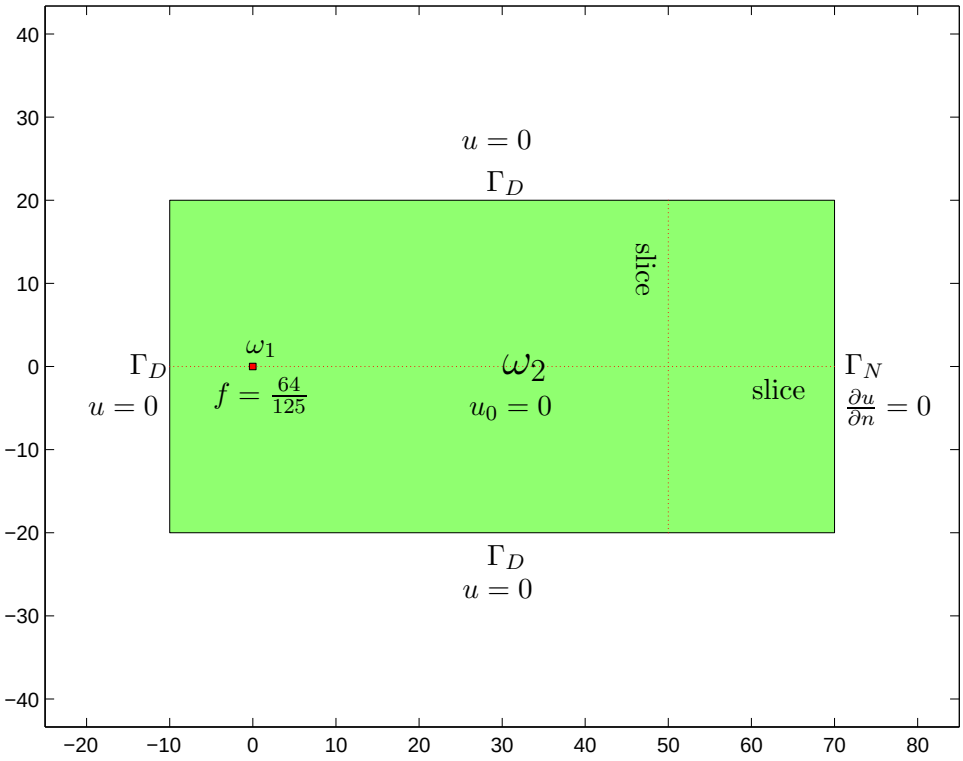


Figure 4.24: Geometry and setting in Example 4.4.

4.4 Pollution in Groundwater Flow

Our final example, taken from Höfinger [12], is concerned with a practical simulation, namely the approximation of the spreading of pollution in groundwater flow:

$$\begin{aligned} \frac{\partial u}{\partial t} - \alpha_D v_p \triangle u + v_p \partial_x u &= \begin{cases} f & \text{in } \omega_1 \times (0,T], \\ 0 & \text{in } \omega_2 \times (0,T], \end{cases} \\ u &= 0 \text{ on } \Gamma_D \times (0,T], \\ \vec{n} \cdot \nabla u &= 0 \text{ on } \Gamma_N \times (0,T] \\ u_0 &= 0 \text{ in } \Omega, \end{aligned} \tag{4.9}$$

where $\Omega := [-10,-20] \times [70,20]$, $\omega_1 := [-0.625,-0.625] \times [0.625,0.625]$, $\omega_2 := \Omega \setminus \omega_1$, $\Gamma_D := \{-10\} \times (-20,20) \cup (-10,70) \times \{-20\} \cup (-10,70) \times \{20\}$, $\Gamma_N := \{70\} \times (-20,20)$. Note that the coefficients α_D, v_p and the source term f are assumed to be constant and the final time $T = 4.36 \cdot 10^6 s \mathrel{\hat{=}} 50$ days.

In Figure 4.24 the setting in this example is shown. Though Ω is two dimensional, we simulate a layer of a porous medium with height h, where the solution u is supposed to be the same for all values in the z direction. We have to take that consideration into account for the units of the coefficients and the source term. The solution u is the concentration of pollution in the groundwater in mg/m^3. We denote the fraction of the medium, that may be filled with fluid by the dimensionless constant n_e, the so-called effective porous volume. The constant $\alpha_D = [m]$ is called diffusivity. The mean velocity of the groundwater through the medium, the pore velocity, is denoted by $v_p = [m/s]$ and is proportional to the speed of the convection. We want the

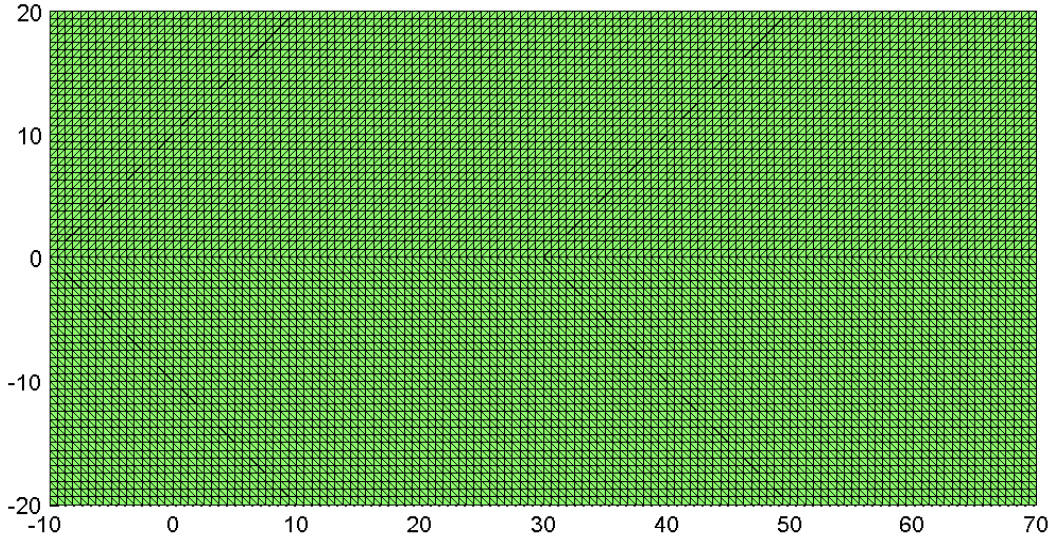


Figure 4.25: Initial triangulation in Example 4.4 with 16384 elements and $|\mathcal{N}_f| = 8064$ free nodes.

pollution source f to give a constant inflow of $M = 2mg/s$. Since we chose $|\omega_1| = (5/4)^2$ the source term yields

$$f = \frac{M}{|\omega_1| h n_e} = \frac{64}{125} mg\, m^{-3}\, s^{-1}.$$

Table 4.8 shows a summary of the constants in this problem as well as the used values. Note that, since $\alpha_D = 0.05$, this problem is convection dominated. With no pollution in Ω at $t = 0$ we expect it to spread towards the right boundary Γ_N.
We only carry out adaptive computations with time discretization parameter $\theta = 1/2$ (Crank-Nicholson) in this example. To resolve the pollution source at $(0, 0)$, we used the initial triangulation from Figure 4.25 with $|\mathcal{N}_f| = 8385$ free nodes. The pollution flow corresponding to the source term f, is switched on instantly at initial time $t = 0$. To simulate this process, we set the time derivative $\partial f/\partial t = 1/t^{3/4}$. This arbitrary choice is a compromise between efficiency and accuracy and gives rise to an additional contribution to the temporal error estimator $\eta_\tau^{(n)}$ for $n = 1, \ldots, N$. In particular, this leads to small time step sizes τ_n used by the adaptive algorithm close to $t = 0$, c.f. Section 3.7.
In Figure 4.26, we plot a projection of the discrete solution $u_{h,\tau}$ at final time $T = 50d$ obtained by an adaptive computation with tolerance $\varepsilon = 8$. We observe, that the pollution spread until approximately $x = 50$ in positive x-direction. The final grid at $T = 50d$ for an analogous computation with tolerance $\varepsilon = 128$ is shown in Figure 4.27.

solution	u	$mg\, m^{-3}$
diffusivity	α_D	$0.05\ m$
pore velocity	v_p	$10^{-5} m\, s^{-1}$
effective porous volume	n_e	0.25
height of the layer	h	$10\, m$
pollution source	f	$64/125\ mg\, m^{-3}\, s^{-1}$

Table 4.8: Summary of the constants and values used in Example 4.4.

Adaptive strategy: Solution after 50 days

We focus on the discrete solution $u_{h,\tau}$ at final time $T = 50d$ for two orthogonal slices $x = 50$ and $y = 0$, indicated in Figure 4.24. To measure the quality of the approximations, we compare the obtained values of $u_{h,\tau}$ on these slices to the analytic solution for an infinite plane and point source at $(0, 0)$ taken from Höfinger [12], which reads

$$u(x, y, t) = \frac{u_0}{4\sqrt{\pi\alpha_D r}} \exp\left(\frac{x-r}{2\alpha_D}\right) \operatorname{erfc}\left(\frac{r - v_p t}{2\sqrt{\alpha_D v_p t}}\right) \text{ with } r = \sqrt{x^2 + y^2} \text{ and } u_0 = \frac{M}{h n_e v_p}. \tag{4.10}$$

Here, $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ denotes the complementary error function. We do not expect global convergence $|||u(T) - u_{h,\tau}(T)||| \to 0$ for $\varepsilon \to 0$. However, for coordinates that are sufficiently far away from the source at $(0, 0)$, we expect that $u_{h,\tau}$ tends towards u.
For tolerances $\varepsilon = 128, \ldots, 8$ we obtain the values of $u_{h,\tau}(50, y, T)$ and $u_{h,\tau}(x, 0, T)$ by adaptive computations. In Figure 4.28, we plot the values $u_{h,\tau}(50, y, T)$ and the corresponding analytic solution $u(50, y, T)$, according to (4.10), over the y-axis. We observe that for smaller ε the discrete solutions approach the analytic solution.
In Figure 4.29, we plot the values $u_{h,\tau}(x, 0, T)$ and the corresponding analytic solution $u(x, 0, T)$, according to (4.10), over the x-axis. The behavior of the analytic solution u in the x interval $[40, 50]$ is dominated by the term with the complementary error function. Again, the approximations become better for smaller ε with regard to the distance to the analytic solution.
To visualize the spatial adaptivity of the algorithm, we plot the number of free nodes $|\mathcal{N}_f^{(n)}|$ for each time step t_n, $n = 1, \ldots, N$, over time between $t = 0$ and final time $T = 50d$ in Figure 4.30. Though there is local coarsening, we observe a general increase of free nodes used by the algorithm. Finally, in Figure 4.31, we plot the time step sizes τ_n for each time step t_{n-1} on the same time interval[3]. Here, we observe that the time step size constantly increases toward the end of the time interval $(0, 50d]$.

[3]For Figure 4.30 and 4.31 the discrete values are linearly interpolated.

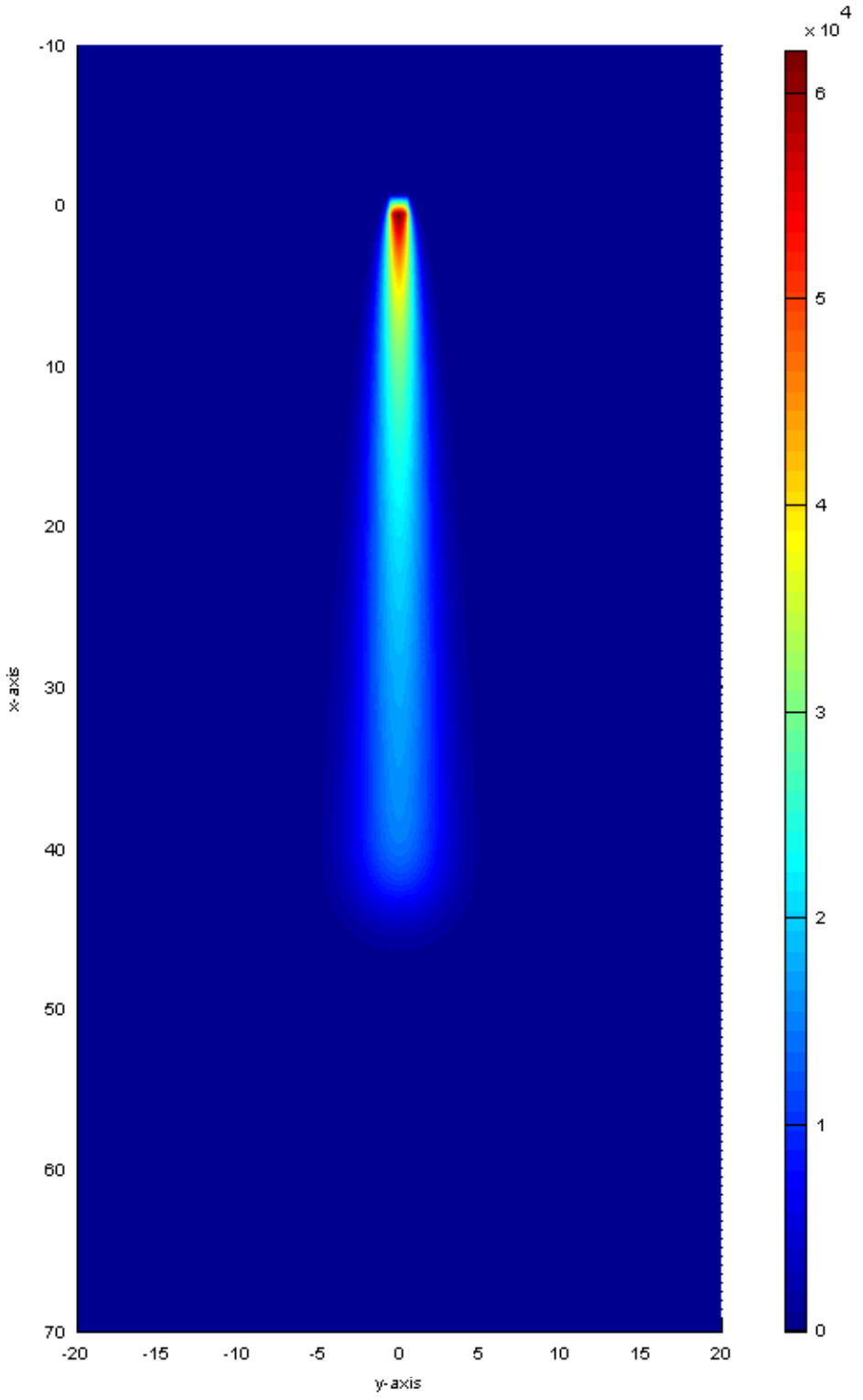


Figure 4.26: Projection of the discrete solution $u_{h,\tau}$ for final time $T = 50d$ for an adaptive computation with time discretization parameter $\theta = 1/2$ and tolerance $\varepsilon = 8$ in Example 4.4 onto the $x - y$ plane. The values of $u_{h,\tau}$ are indicated by the color bar on the right hand side.

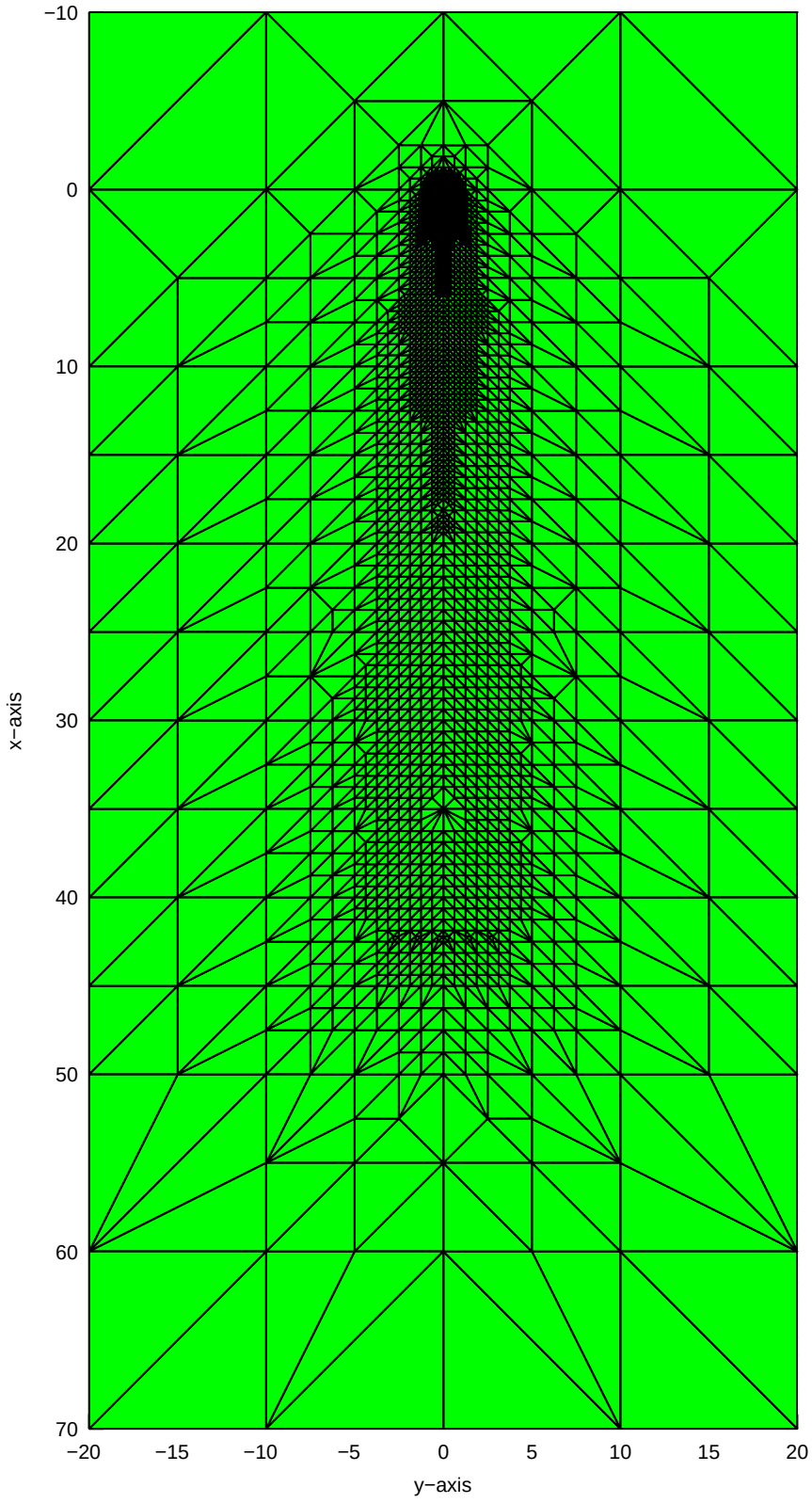


Figure 4.27: Final grid for time $T = 50d$ for an adaptive computation with time discretization parameter $\theta = 1/2$ and tolerance $\varepsilon = 128$ in Example 4.4.

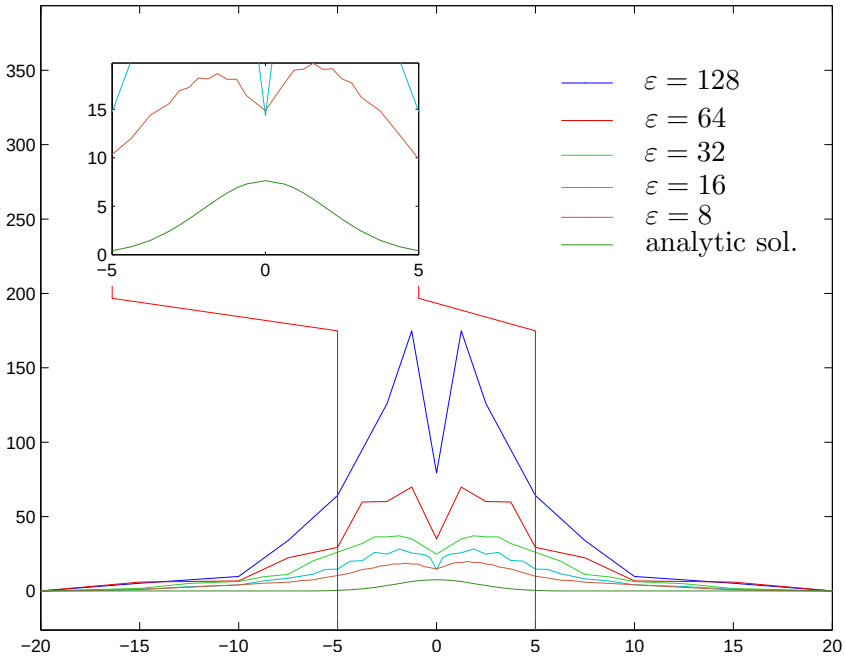


Figure 4.28: Discrete solution $u_{h,\tau}(50, y, T)$ at final time $T = 50d$ obtained from adaptive computations with time discretization parameter $\theta = 1/2$ in Example 4.4. We plot the values of $u_{h,\tau}$ over the y-axis with $y \in [-20, 20]$. Moreover, we add the corresponding analytic solution $u(50, y, T)$ from (4.10). We observe that the approximations, obtained by computations with tolerance $\varepsilon = 128, \ldots, 8$, approach the discrete solution.

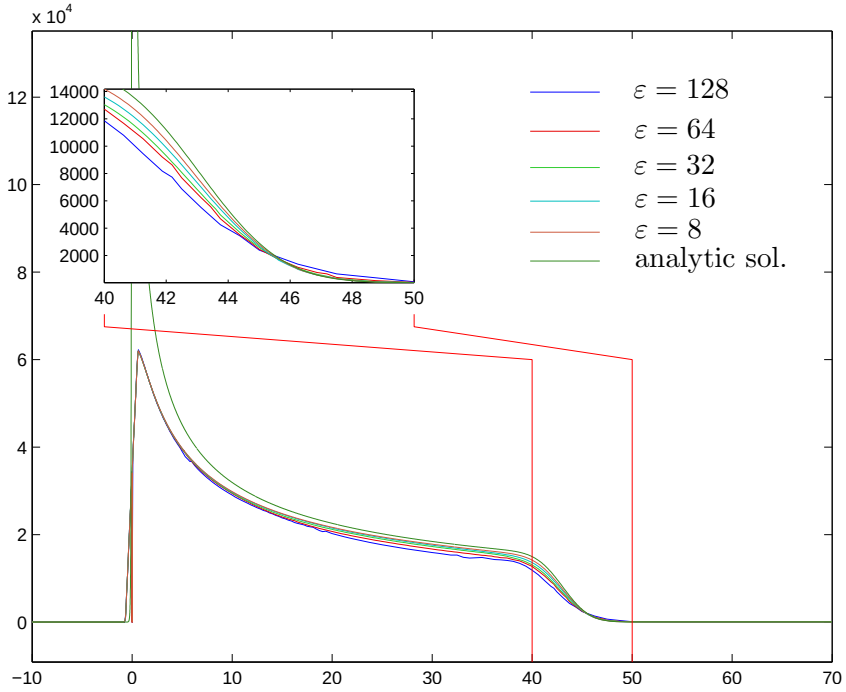


Figure 4.29: Discrete solution $u_{h,\tau}(x, 0, T)$ at final time $T = 50d$ obtained from adaptive computations with time discretization parameter $\theta = 1/2$ in Example 4.4. We plot the values of $u_{h,\tau}$ over the x-axis with $x \in [-10, 70]$. Moreover, we add the corresponding analytic solution $u(x, 0, T)$ from (4.10). We observe that the approximations, obtained by computations with tolerance $\varepsilon = 128, \ldots, 8$, approach the discrete solution.

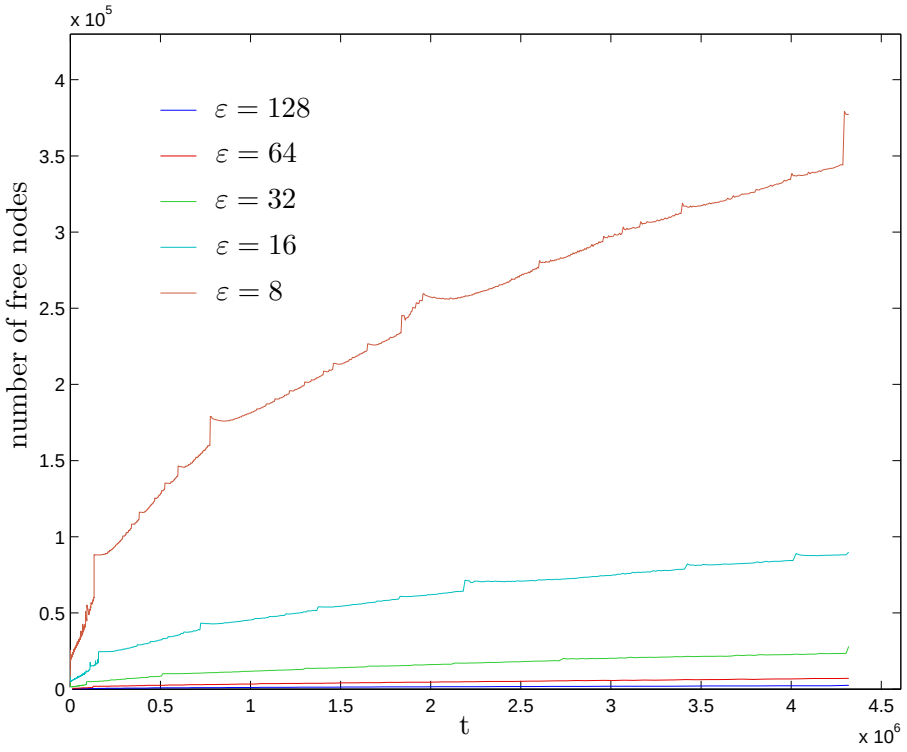


Figure 4.30: Number of free nodes $|\mathcal{N}_f^{(n)}|$ for each time step t_n in Example 4.4 between $t = 0$ and final time $T = 50d$, used by the adaptive algorithm with time discretization parameter $\theta = 1/2$ and tolerances $\varepsilon \in \{128, \ldots, 8\}$. For visualization the discrete values of $|\mathcal{N}_f^{(n)}|$ are linearly interpolated.

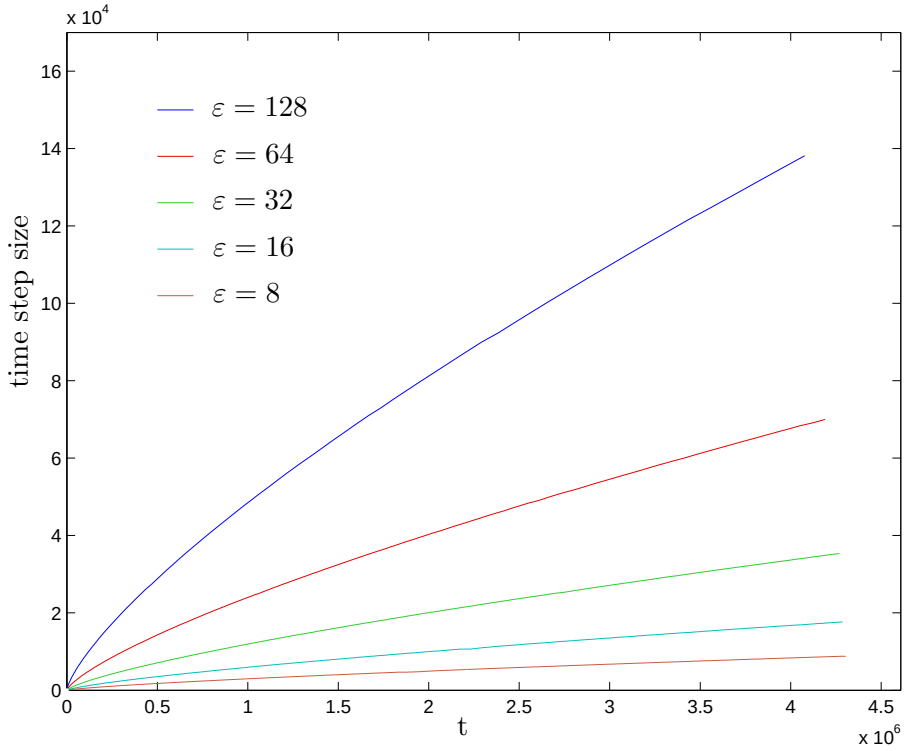


Figure 4.31: Time step sizes τ_n for each time step t_{n-1} in Example 4.4 between $t = 0$ and final time $T = 50d$, used by the adaptive algorithm with time discretization parameter $\theta = 1/2$ and tolerances $\varepsilon \in \{128, \ldots, 8\}$. For visualization the discrete values of τ_n are linearly interpolated and we do not plot the final time step size τ_N.

Appendix A

Codes

A.1 The Function *refineGrid*

```
0001 function [Grid,LogStruct] = refineGrid(Grid,MarkedElements,Driver)      %ok
0002 %   refineGrid: Red-refines all marked triangles for a given grid,
0003 %               then green-completion
0004 %
0005 %   INPUT ARGUMENTS
0006 %
0007 %   Grid: structure with fields C4N,N4E,N4D,N4N,F4E,NOE,Unused,M,H,b,
0008 %                               DoubleArea
0009 %   [C4N: Coordinates;a nx2-matrix, where n is the number of vertices
0010 %       in the grid.
0011 %       Format of ith row: 1st coordinate of ith vertex/2nd coordinate of
0012 %       ith vertex
0013 %
0014 %   [N4E: Elements;nx5-matrix, where n is the number of triangles in
0015 %       the grid.
0016 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0017 %       triangle/3rd vertex of ith triangle/father(referring to F4E-Matrix)
0018 %       /green brother]
0019 %
0020 %   [N4D: Dirichlet nodes; nx2-matrix, where n is the number of
0021 %       dirichlet edges.
0022 %       Format of ith row: 1st vertex of ith dirichlet edge/2nd vertex of
0023 %       ith dirichlet edge]
0024 %
0025 %   [N4N: Neumann nodes; see Dirichlet]
0026 %
0027 %   [F4E: Fathers; nx8-matrix, where n is the number of fathers in
0028 %       the grid. A father is a triangle which is split in 4 congruent
0029 %       subtriangles(called children) given by bisecting all edges of the
0030 %       father. A father is not in the actual Element-matrix, since it is
0031 %       not an actual element. His children may, but don't have to be in
0032 %       this matrix, they too can be splitted and may be in the
0033 %       Fathers-matrix;
0034 %       Format of ith row: 1st vertex of ith father/2nd vertex of ith
```

```
0035 %       father/3rd vertex of ith father/middle child/2nd child/3rd child
0036 %       /4th child/father of father
0037 %       Remark: column 1-3 referring to C4N-matrix
0038 %               column 4-7 referring to N4E-matrix
0039 %               column 8 referring to F4E-matrix ]
0040 %
0041 %   [NON: Nodes on Edges; nxn-matrix, where n is number of vertices.
0042 %       With NON the algorithm administers vertices (normal and
0043 %       hanging ones). In ith row/jth column there is the number of the
0044 %       vertex on the edge between the ith and jth vertice(referring to
0045 %       C4N-matrix) if there is none or node lies on a boundary edge
0046 %       the entry is 0]
0047 %
0048 %   [Unused.C4N/N4E/F4E: Unused rows from these matrices;
0049 %       Format: vertical vector with row number entries
0050 %
0051 %   [M,H,b,DoubleArea: Saved mass-matrix,
0052 %       transport-matrix, rhs and double areas for Grid. These fields are
0053 %       deleted, if a refinement process is conducted
0054 %
0055 %   [MarkedElements: nx2 matrix, defines which of the Elements are to be
0056 %       refined (1st column); if MarkedElements=='All'->uniform refinement]
0057 %
0058 %   [Driver: name of the file, where the problem data is located; string]
0059 %
0060 %   OUTPUT ARGUMENTS:
0061 %
0062 %   [Grid: see input]
0063 %
0064 %   [LogStruct is a log-struct which contains information about the
0065 %       solution process.]
0066 %
0067 %   See also coarsenGrid
0068 %
0069 %   Copyright (c) 2007 Philipp Wissgott
0070 %                      Vienna University of Technology
0071
0072 %************************** OPTIONS ************************************
0073
0074   CPU_time = cputime;
0075   OptionFile = feval(Driver,'OptionFile');
0076   Options = feval(OptionFile,'LoadOptions','Refinement',Driver,0,1);
0077
0078 %************************** TO LOG, OR NOT TO LOG **********************
0079
0080   if nargout == 2
0081
0082      Options.Logging = 1;
0083      %Log the size of input matrices
```

```
0084      LogStruct.PreSize.C4N = size(Grid.C4N,1);
0085      LogStruct.PreSize.N4E = ...
0086                          size(setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E),2);
0087      LogStruct.PreSize.N4D = size(Grid.N4D,1);
0088      LogStruct.PreSize.N4N = size(Grid.N4N,1);
0089   else
0090      Options.Logging = 0;
0091   end
0092
0093   %input for the recursive call, default value
0094   LogStruct.AdditionalMarkedElements = [];
0095
0096 %************************** CHECK INPUT ********************************
0097
0098   if isa(MarkedElements,'char') && strcmp(MarkedElements,'All')
0099       %Uniform refinement
0100       MarkedElements = zeros(size(Grid.N4E,1),2);
0101       MarkedElements(:,1) = ones(size(Grid.N4E,1),1);
0102   end
0103
0104   if ~nnz(MarkedElements(:,1))
0105       if Options.Logging
0106          LogStruct.PostSize = LogStruct.PreSize;
0107          LogStruct.Info = ...
0108              'refineNet-Info : No elements marked in net -> no refinement';
0109       end
0110       LogStruct.AdditionalMarkedElements = [];
0111       return;
0112   elseif size(MarkedElements,1)<size(Grid.N4E,1)
0113       %function expects nx2 matrix, fill up
0114       MarkedElements = ...
0115          [MarkedElements;zeros(size(Grid.N4E,1)-size(MarkedElements,1),2)];
0116   end
0117
0118   if nnz(Grid.N4E(:,5))
0119      [Grid.N4E,MarkedElements] = ...
0120                             greenReverseCompletion(Grid.N4E,MarkedElements);
0121   end
0122
0123
0124   if ~isempty(Grid.N4E)
0125       nze = setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E);
0126   else
0127       error('Elements-matrix empty!');
0128   end
0129
0130   if isempty(Grid.C4N)
0131       error('Coordinates-matrix empty!');
0132   end
```

```
0133
0134   if isempty(Grid.N4D)
0135       error('Dirichlet-matrix empty!');
0136   end
0137
0138   if isempty(Grid.NOE)
0139       if Options.Logging
0140          LogStruct.Info = ...
0141           'refineNet-Info : Using default NodesOnEdges->no hanging nodes';
0142       end
0143       Grid.NOE = sparse(size(Grid.C4N,1),size(Grid.C4N,1));  %default
0144   end
0145
0146   %Delete saved data from Grid
0147   Grid.M = [];
0148   Grid.H = [];
0149   Grid.b = [];
0150   Grid.DoubleArea = [];
0151
0152 %***************************** MAIN ************************************
0153
0154   %Red-refinement
0155   [Grid,LogStruct] = refineElements(Grid,MarkedElements,LogStruct);
0156
0157   %Completion if necessary
0158   if nnz(Grid.NOE-Grid.NOE')
0159       Grid.N4E = greenCompletion(Grid.N4E,Grid.NOE,Grid.Unused.N4E);
0160   end
0161
0162  %Finalize logging
0163  if Options.SaveOutput
0164     save refinementOutput
0165  end
0166
0167  if Options.Logging
0168      LogStruct.PostSize.C4N = size(Grid.C4N,1);
0169      LogStruct.PostSize.N4E = size(find(Grid.N4E(:,1)),1);
0170      LogStruct.PostSize.N4D = size(Grid.N4D,1);
0171      LogStruct.PostSize.N4N = size(Grid.N4N,1);
0172      LogStruct.ComputationTime = cputime - CPU_time;
0173  end
0174
0175  if Options.DisplayOutput
0176     %graphical representation
0177     show(Grid,zeros(size(Grid.C4N,1),1),'Refined grid',1)
0178     view(0,90);
0179  end
0180
0181  if Options.ShowTime
```

```
0182        ComputationTimeRefinement = cputime - CPU_time
0183    end
0184
0185 %end of refineGrid
0186 %-------------------------------------------------------------------------
0187 function [Grid,LogStruct] =...
0188             refineElements(Grid,MarkedElements,LogStruct)
0189 %   refineElements: Red-refines all marked triangles in a given grid,
0190 %                   recursion
0191
0192 %*************************** RENAMING FOR READABILITY ********************
0193
0194   C4N = Grid.C4N;
0195   N4E = Grid.N4E;
0196   N4D = Grid.N4D;
0197   try
0198       N4N = Grid.N4N;
0199   catch
0200       N4N = [];
0201   end
0202   F4E = Grid.F4E;
0203   NOE = Grid.NOE;
0204
0205 %***************************** MAIN ************************************
0206
0207   %Initialization
0208   MarkedElements = find(MarkedElements(:,1));
0209   LogStruct.AdditionalMarkedElements = [];
0210
0211   while ~isempty(MarkedElements)
0212
0213       nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0214       mnze = intersect(MarkedElements,nze); %marked non zero elements
0215       %Assemble ElementsOverEdge needed to get ElementsToCheck, i.e. the
0216       %elements which are checked after red-refinement if they have more
0217       %than one hanging node
0218       a = [N4E(nze,1) N4E(nze,2) N4E(nze,3)]';
0219       I1 = reshape(a,3*nnz(nze),1);
0220       a = [N4E(nze,2) N4E(nze,3) N4E(nze,1)]';
0221       I2 = reshape(a,3*nnz(nze),1);
0222       a = [nze; nze; nze];
0223       Values = reshape(a,3*nnz(nze),1);
0224       ElementsOverEdge = sparse(I1,I2,Values,size(C4N,1),size(C4N,1));
0225       ElementsToCheck = zeros(1,6*nnz(mnze));
0226       for j=1:size(mnze,2)
0227           Nodes = N4E(mnze(j),1:3);
0228           if N4E(mnze(j),4)
0229              FatherNodes = F4E(N4E(mnze(j),4),1:3);
0230           end
```

```
0231            for i=1:3
0232               ElementsToCheck((j-1)*6+i) = ...
0233                              ElementsOverEdge(Nodes(mod(i,3)+1),Nodes(i));
0234               if N4E(mnze(j),4)
0235                  ElementsToCheck((j-1)*6+3+i) = ...
0236                   ElementsOverEdge(FatherNodes(mod(i,3)+1),FatherNodes(i));
0237               end
0238            end
0239         end
0240         ElementsToCheck = setdiff(ElementsToCheck,0);
0241
0242         %prelocating memory for new nodes with arbitrary value -1
0243         I1 = [N4E(mnze,1); N4E(mnze,2); N4E(mnze,3)];
0244         I2 = [N4E(mnze,2); N4E(mnze,3); N4E(mnze,1)];
0245         NOE = NOE + sparse(I1,I2,-1,size(NOE,1),size(NOE,1));
0246         RealSizeC4N = size(C4N,1); %tracks the current size of C4N
0247         C4N = [C4N; ones(max(3*nnz(mnze)-nnz(Grid.Unused.C4N),0),2)];
0248         RealSizeN4E = size(N4E,1); %tracks the current size of C4N
0249         N4E = [N4E; ones(max(3*nnz(mnze)-nnz(Grid.Unused.N4E),0),5)];
0250         RealSizeF4E = size(F4E,1); %tracks the current size of C4N
0251         F4E = [F4E; ones(max(nnz(mnze)-nnz(Grid.Unused.F4E),0),8)];
0252         Children = zeros(4,5);
0253         NewVertices = zeros(1,3);
0254
0255         MarkedElements = []; %marked elements due to grid condition
0256
0257         for j=1:size(mnze,2)
0258
0259             LocElement = N4E(mnze(j),1:4);
0260             FatherOfElement = LocElement(4);
0261             ElementEdges = [LocElement([1 2 3])',LocElement([2 3 1])'];
0262
0263             for n=1:3
0264                 if NOE(ElementEdges(n,2),ElementEdges(n,1))>0
0265                 %there is already a vertex on this edge
0266                     NewVertices(n) = ...
0267                                    NOE(ElementEdges(n,2),ElementEdges(n,1));
0268                     NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0269                              NOE(ElementEdges(n,2),ElementEdges(n,1));
0270                 else
0271                     %create new vertex
0272                     if isempty(Grid.Unused.C4N)
0273                         NewVertices(n) = RealSizeC4N + 1;
0274                         NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0275                                                 NewVertices(n); %update NOE
0276                         C4N(RealSizeC4N+1,:) = (C4N(ElementEdges(n,1),:) ...
0277                                               + C4N(ElementEdges(n,2),:))/2;
0278                         RealSizeC4N = RealSizeC4N + 1; %update size of C4N
0279                     else
```

```
0280                    C4N(Grid.Unused.C4N(1),:) = ...
0281                                        (C4N(ElementEdges(n,1),:) ...
0282                                          + C4N(ElementEdges(n,2),:))/2;
0283                    NewVertices(n) = Grid.Unused.C4N(1);
0284                    NOE(ElementEdges(n,1),ElementEdges(n,2)) = ...
0285                                                    Grid.Unused.C4N(1);
0286                    Grid.Unused.C4N = ...
0287                              Grid.Unused.C4N(2:size(Grid.Unused.C4N,1));
0288                end
0289            end
0290        end
0291        %create new elements
0292        if isempty(Grid.Unused.N4E)
0293            NewElements = [RealSizeN4E+1:RealSizeN4E+3]';
0294            RealSizeN4E = RealSizeN4E + 3;
0295        else
0296            NewElements = Grid.Unused.N4E(1:3);
0297            Grid.Unused.N4E = Grid.Unused.N4E(4:size(Grid.Unused.N4E,1));
0298        end
0299        if isempty(Grid.Unused.F4E)
0300            NewFather = RealSizeF4E + 1;
0301            RealSizeF4E = RealSizeF4E + 1;
0302        else
0303            NewFather = Grid.Unused.F4E(1);
0304            Grid.Unused.F4E = Grid.Unused.F4E(2:size(Grid.Unused.F4E,1));
0305        end
0306        Children(1,1:3) = NewVertices;
0307        Children(2:4,1:3) = diag(LocElement(1:3));
0308        Children(3,1) = NewVertices(1);
0309        Children(2,2) = NewVertices(1);
0310        Children(2,3) = NewVertices(3);
0311        Children(4,1) = NewVertices(3);
0312        Children(3,3) = NewVertices(2);
0313        Children(4,2) = NewVertices(2);
0314        Children(:,4) = NewFather;
0315        N4E(mnze(j),:) = Children(1,:);
0316        N4E(NewElements,:) = Children(2:4,:);
0317        F4E(NewFather,1:3) = LocElement(1:3);
0318        F4E(NewFather,4) = mnze(j);
0319        F4E(NewFather,5:7) = NewElements';
0320        F4E(NewFather,8) = FatherOfElement;
0321    end
0322
0323    %finalize sizes
0324    C4N = C4N(1:RealSizeC4N,:);
0325    F4E = F4E(1:RealSizeF4E,:);
0326    N4E = N4E(1:RealSizeN4E,:);
0327
0328    %update NOE size
```

```
0329        NOE = [NOE sparse(size(NOE,1),size(C4N,1)-size(NOE,1));...
0330          sparse(size(C4N,1)-size(NOE,1),size(C4N,1))];
0331
0332        %Update Dirichlet edges
0333        NewDirichlet = [];
0334        for j=1:size(N4D,1)
0335           if NOE(N4D(j,1),N4D(j,2))
0336               NewDirichlet = [NewDirichlet; ...
0337                               N4D(j,1) NOE(N4D(j,1),N4D(j,2)); ...
0338                               NOE(N4D(j,1),N4D(j,2)) N4D(j,2)];
0339               NOE(N4D(j,1),N4D(j,2)) = 0;
0340               NOE(N4D(j,2),N4D(j,1)) = 0;
0341           else
0342               NewDirichlet =[NewDirichlet; N4D(j,:)];
0343           end
0344        end
0345        N4D = NewDirichlet;
0346
0347        %Update Neumann edges
0348        NewNeumann = [];
0349        for j=1:size(N4N,1)
0350           if NOE(N4N(j,1),N4N(j,2))
0351               NewNeumann = [NewNeumann; N4N(j,1) NOE(N4N(j,1),N4N(j,2)); ...
0352                               NOE(N4N(j,1),N4N(j,2)) N4N(j,2)];
0353               NOE(N4N(j,1),N4N(j,2)) = 0;
0354               NOE(N4N(j,2),N4N(j,1)) = 0;
0355           else
0356               NewNeumann =[NewNeumann; N4N(j,:)];
0357           end
0358        end
0359        N4N = NewNeumann;
0360
0361        %Because of new elements, grid condition may be violated
0362        nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0363        HangingNodes = abs(NOE-NOE');
0364
0365        if nnz(HangingNodes)
0366
0367            %Check only elements which have been identified via
0368            %ElementsToCheck
0369            for j=1:size(ElementsToCheck,2)
0370                LocElement = N4E(ElementsToCheck(j),1:3);
0371                ElementEdges = [LocElement([1 2 3])',LocElement([2 3 1])'];
0372                HangingEdgesCounter = 0;
0373                for n=1:3
0374                    if HangingNodes(ElementEdges(n,2),ElementEdges(n,1))
0375                       HangingNode = ...
0376                           HangingNodes(ElementEdges(n,2),ElementEdges(n,1));
0377                       if HangingNodes(ElementEdges(n,2),HangingNode) ...
```

```
0378                        || HangingNodes(HangingNode,ElementEdges(n,1))
0379                        MarkedElements = ...
0380                                    [MarkedElements; ElementsToCheck(j)];
0381                        break;
0382                    else
0383                        HangingEdgesCounter = HangingEdgesCounter + 1;
0384                    end
0385                end
0386            end
0387            if HangingEdgesCounter>1
0388                MarkedElements = [MarkedElements; ElementsToCheck(j)];
0389            end
0390        end
0391    end
0392
0393    %delete prelocation values
0394    [I1,I2] = find(NOE==-1);
0395    NOE = NOE + sparse(I1,I2,1,size(NOE,1),size(NOE,1));
0396
0397    %prepare for next run
0398    MarkedElements = unique(MarkedElements);
0399    LogStruct.AdditionalMarkedElements = ...
0400                    [LogStruct.AdditionalMarkedElements; MarkedElements];
0401
0402  end
0403
0404  %Back to original data structures
0405  Grid.C4N = C4N;
0406  Grid.N4E = N4E;
0407  Grid.N4D = N4D;
0408  Grid.N4N = N4N;
0409  Grid.F4E = F4E;
0410  Grid.NOE = NOE;
0411
0412  %end of refineElements
0413 %--------------------------------------------------------------------------
0414 function N4E = greenCompletion(N4E,NOE,UnusedElements)
0415 %greenCompletion: green-refine all elements of N4E with a hanging node
0416
0417  nze = setdiff(1:size(N4E,1),UnusedElements);
0418  Nnze = length(nze);
0419  HangingNodes = abs(NOE-NOE');
0420  a = N4E(nze,1:3)';
0421  I1 = reshape(a,3*Nnze,1);
0422  a = N4E(nze,[2 3 1])';
0423  I2 = reshape(a,3*Nnze,1);
0424  a = repmat(1:Nnze,3,1);
0425  Val = reshape(a,3*Nnze,1);
0426  ElementsOnEdges = sparse(I1,I2,Val,size(NOE,1),size(NOE,2));
```

```
0427   [I1,I2] = find(HangingNodes);
0428   [I1,I2,ElementsToConsider] = ...
0429             find(sparse(I1,I2,1,size(NOE,1),size(NOE,2)).*ElementsOnEdges);
0430   ElementsToConsider = sort(nze(ElementsToConsider));
0431   %prelocation
0432   Counter = size(N4E,1);
0433   N4E = [N4E; zeros(nnz(HangingNodes)/2,5)];
0434
0435
0436   for j=1:nnz(ElementsToConsider)
0437       Element = ElementsToConsider(j);
0438       %EdgesOfElement = [N4E(nze(j),[1 2 3])' N4E(nze(j),[2 3 1])'];
0439       for k=1:3
0440            if HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k))
0441               HangingNode = ...
0442                       HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k));
0443               Counter = Counter + 1;
0444               LocElement = N4E(Element,:);
0445               N4E(Element,:) = ...
0446                      [HangingNode LocElement(rem(k+1,3)+1) LocElement(k)...
0447                                                    LocElement(4) Counter];
0448               N4E(Counter,:) = [HangingNode LocElement(rem(k,3)+1)...
0449                           LocElement(rem(k+1,3)+1) LocElement(4) Element];
0450
0451               break;
0452           end
0453
0454       end
0455   end
0456
0457   N4E = N4E(1:Counter,:);
0458
0459  %end of greenCompletion
0460 %--------------------------------------------------------------------------
0461 function [N4E,MarkedElements] = greenReverseCompletion(N4E,MarkedElements)
0462 %greenReverseCompletion: reverse green completion
0463
0464   GreenElements = find(N4E(:,5));
0465   for j=1:size(GreenElements,1)
0466
0467       GreenBrother = N4E(GreenElements(j),5);
0468
0469       %Stop when coming to lastborns
0470       if GreenBrother<GreenElements(j)
0471           N4E = N4E(1:GreenElements(j)-1,:);
0472           MarkedElements = MarkedElements(1:GreenElements(j)-1,:);
0473           return;
0474       end
0475
```

```
0476        %Reversing green splitting
0477        ThirdNode =  setdiff([N4E(GreenBrother,2:3)],...
0478                             [N4E(GreenElements(j),2:3)]);
0479        N4E(GreenElements(j),1:3) = [N4E(GreenElements(j),2)...
0480                                     N4E(GreenElements(j),3) ThirdNode];
0481        N4E(GreenElements(j),5) = 0;
0482
0483        %Update MarkedElements
0484        if MarkedElements(GreenBrother,1) %offensive refinement
0485              MarkedElements(GreenElements(j),1) = 1;
0486        end
0487
0488        if MarkedElements(GreenElements(j),2) ...%defensive coarsening
0489           && ~MarkedElements(GreenBrother,2)
0490            MarkedElements(GreenElements(j),2) = 0;
0491        end
0492
0493    end
0494
0495   %end of greenReverseCompletion
```

A.2 The Function *coarsenGrid*

```
0001 function [Grid,LogStruct] = coarsenGrid(Grid,MarkedElements,Driver)    %ok
0002 %   coarsenGrid: Coarsens marked triangles in a given grid, then
0003 %   green completion
0004 %
0005 %   INPUT ARGUMENTS
0006 %
0007 %   Grid: structure with fields C4N,N4E,N4D,N4N,F4E,NOE,Unused,M,H,b,
0008 %                               DoubleArea
0009 %   [C4N: Coordinates;a nx2-matrix, where n is the number of vertices
0010 %       in the grid.
0011 %       Format of ith row: 1st coordinate of ith vertex/2nd coordinate of
0012 %       ith vertex
0013 %
0014 %   [N4E: Elements;nx5-matrix, where n is the number of triangles in
0015 %       the grid.
0016 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0017 %       triangle/3rd vertex of ith triangle/father(referring to F4E-Matrix)
0018 %       /green brother]
0019 %
0020 %   [N4D: Dirichlet nodes; nx2-matrix, where n is the number of
0021 %       dirichlet edges.
0022 %       Format of ith row: 1st vertex of ith dirichlet edge/2nd vertex of
0023 %       ith dirichlet edge]
0024 %
0025 %   [N4N: Neumann nodes; see N4D]
0026 %
```

```
0027 %   [F4E: Fathers; nx8-matrix, where n is the number of fathers in
0028 %       the grid. A father is a triangle which is split in 4 congruent
0029 %       subtriangles(called children) given by bisecting all edges of the
0030 %       father. A father is not in the actual Element-matrix, since it is
0031 %       not an actual element. His children may, but don't have to be in
0032 %       this matrix, they too can be splitted and may be in the
0033 %       Fathers-matrix;
0034 %       Format of ith row: 1st vertex of ith father/2nd vertex of ith
0035 %       father/3rd vertex of ith father/middle child/2nd child/3rd child
0036 %       /4th child/father of father
0037 %       Remark: column 1-3 referring to C4N-matrix
0038 %               column 4-7 referring to N4E-matrix
0039 %               column 8 referring to F4E-matrix ]
0040 %
0041 %   [NON: Nodes on Edges; nxn-matrix, where n is number of vertices.
0042 %       With NON the algorithm administers vertices (normal and
0043 %       hanging ones). In ith row/jth column there is the number of the
0044 %       vertex on the edge between the ith and jth vertice(referring to
0045 %       C4N-matrix) if there is none or node lies on a boundary edge
0046 %       the entry is 0]
0047 %
0048 %   [Unused.C4N/N4E/F4E: Unused rows from these matrices;
0049 %       Format: vertical vector with row number entries
0050 %
0051 %   [M,H,b,DoubleArea: Saved mass-matrix,
0052 %       transport-matrix, rhs and double areas for Grid. These fields are
0053 %       deleted, if a coarsening process is conducted
0054 %
0055 %   [MarkedElements: nx2 matrix, defines which of the Elements are to be
0056 %       coarsened (2nd column);
0057 %       if MarkedElements=='All'->uniform coarsening]
0058 %
0059 %   [Driver: name of the file, where the problem data is located; string]
0060 %
0061 %   OUTPUT ARGUMENTS:
0062 %
0063 %   [Grid: see input]
0064 %
0065 %   [LogStruct is a log-struct which contains information about the
0066 %    solution process.]
0067 %
0068 %
0069 %   See also refineGrid
0070 %
0071 %   Copyright (c) 2007 Philipp Wissgott
0072 %                      Vienna University of Technology
0073
0074 %************************** OPTIONS*************************************
0075
```

```
0076   CPU_time = cputime;
0077   OptionFile = feval(Driver,'OptionFile');
0078   Options = feval(OptionFile,'LoadOptions','Coarsening',Driver,0,1);
0079   Options.CoarseningLevel = ...
0080                             feval(OptionFile,'Coarsening','CoarseningLevel');
0081
0082 %************************** TO LOG, OR NOT TO LOG ***********************
0083
0084   if nargout == 2
0085
0086      Options.Logging = 1;
0087      %Log the size of the input matrices
0088      LogStruct.PreSize.C4N = size(Grid.C4N,1);
0089      LogStruct.PreSize.N4E = ...
0090                          size(setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E),2);
0091      LogStruct.PreSize.N4D = size(Grid.N4D,1);
0092      LogStruct.PreSize.N4N = size(Grid.N4N,1);
0093   else
0094      Options.Logging = 0;
0095   end
0096
0097 %************************** CHECK INPUT*********************************
0098
0099   if strcmp(Options.CoarseningLevel,'Uniform')
0100       MarkedElements = 'All';
0101   end
0102
0103   if isa(MarkedElements,'char') &&  strcmp(MarkedElements,'All')
0104       %Uniform coarsening
0105       MarkedElements = zeros(size(Grid.N4E,1),2);
0106       MarkedElements(:,2) = ones(size(Grid.N4E,1),1);
0107   end
0108
0109   if ~nnz(MarkedElements(:,2))
0110       if Options.Logging
0111           LogStruct.PostSize = LogStruct.PreSize;
0112           LogStruct.Info = ...
0113             'coarsenNet-Info : No elements marked in net -> no coarsening';
0114       end
0115       return;
0116   elseif size(MarkedElements,1)<size(Grid.N4E,1)
0117       MarkedElements = ...
0118          [MarkedElements;zeros(size(Grid.N4E,1)-size(MarkedElements,1),2)];
0119   end
0120
0121   if isempty(Grid.F4E)
0122       if Options.Logging
0123          LogStruct.PostSize = LogStruct.PreSize;
0124          LogStruct.Info = ...
```

```
0125            'coarsenNet-Info : No refined elements in net -> no coarsening';
0126        end
0127        return;
0128    else
0129        nzf = setdiff(1:size(Grid.F4E,1),Grid.Unused.F4E);
0130    end
0131
0132    if ~isempty(Grid.N4E)
0133        if size(Grid.N4E,2)<5
0134            error('Grid.N4E dimension to small');
0135        end;
0136        if nnz(Grid.N4E(:,5))
0137           [Grid.N4E,MarkedElements] = ...
0138                              greenReverseCompletion(Grid.N4E,MarkedElements);
0139        end
0140        nze = setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E);
0141    else
0142        error('Grid.N4E-matrix empty!');
0143    end
0144
0145    if isempty(Grid.C4N)
0146        error('Grid.C4N-matrix empty!');
0147    end
0148
0149    if isempty(Grid.N4D)
0150        error('Grid.N4D-matrix empty!');
0151    end
0152
0153    if isempty(Grid.NOE)
0154        if Options.Logging
0155           LogStruct.Info = ...
0156                'coarsenNet-Info : Using default Grid.NON->no hanging nodes';
0157        end
0158        Grid.NON = sparse(size(Grid.C4N,1),size(Grid.C4N,1));  %default
0159    end
0160
0161    %Delete saved data from Grid
0162    Grid.M = [];
0163    Grid.H = [];
0164    Grid.b = [];
0165    Grid.DoubleArea = [];
0166
0167 %****************************** MAIN ************************************
0168
0169    %Initialization
0170    HigherGenerationFathers = unique(Grid.F4E(find(Grid.F4E(:,8)),8));
0171    LastGenerationFathers = setdiff(nzf,HigherGenerationFathers);
0172    Children = Grid.F4E(LastGenerationFathers,4:7);
0173    %marked last generation fathers, i.e the last generation fathers where
```

```
0174   %all children are marked
0175   mlgf = find(sum([MarkedElements(Children(:,1),2),...
0176                    MarkedElements(Children(:,2),2),...
0177                    MarkedElements(Children(:,3),2),...
0178                    MarkedElements(Children(:,4),2)],2)==4);
0179
0180   for j=1:size(mlgf,1)
0181         Grid = coarsenFather(LastGenerationFathers(mlgf(j)),Grid);
0182   end
0183
0184   %Update edges
0185   [a,b] = find(Grid.NOE==-1);
0186   c = unique(a);
0187   if a
0188       BoundaryJumps = [];
0189   else
0190       BoundaryJumps = 0;
0191   end
0192   for j=1:nnz(c)
0193     d = find(Grid.NOE(c(j),:)==-1);
0194     BoundaryJumps = [BoundaryJumps; c(j) min(d) nnz(d)];
0195   end
0196
0197   %Delete boundary values
0198   [I1,I2] = find(Grid.NOE==-1);
0199   Grid.NOE = Grid.NOE + sparse(I1,I2,1,size(Grid.NOE,1),size(Grid.NOE,1));
0200
0201   %Update Dirichlet edges
0202   NewDirichlet = [];
0203   j=1;
0204   while j<=size(Grid.N4D,1)
0205           d = find(BoundaryJumps(:,1)==Grid.N4D(j,1));
0206       if d
0207           NewDirichlet = [NewDirichlet; ...
0208                           BoundaryJumps(d,1) BoundaryJumps(d,2)];
0209           j = j + BoundaryJumps(d,3) + 1;
0210       else
0211           NewDirichlet = [NewDirichlet; Grid.N4D(j,1) Grid.N4D(j,2)];
0212           j = j + 1;
0213       end
0214   end
0215   a = setdiff(Grid.N4D(:,1),NewDirichlet(:,1));
0216   Grid.Unused.C4N = [Grid.Unused.C4N; a];
0217   Grid.C4N(a,:) = zeros(size(a,1),2);
0218   Grid.N4D = NewDirichlet;
0219
0220   %Update Neumann edges
0221   if ~isempty(Grid.N4N)
0222      NewNeumann = [];
```

```
0223      j=1;
0224      while j<=size(Grid.N4N,1)
0225              d = find(BoundaryJumps(:,1)==Grid.N4N(j,1));
0226         if d
0227             NewNeumann = [NewNeumann; ...
0228                           BoundaryJumps(d,1) BoundaryJumps(d,2)];
0229             j = j + BoundaryJumps(d,3) + 1;
0230         else
0231             NewNeumann = [NewNeumann; Grid.N4N(j,1) Grid.N4N(j,2)];
0232             j = j + 1;
0233         end
0234      end
0235      a = setdiff(Grid.N4N(:,1),NewNeumann(:,1));
0236      Grid.Unused.C4N = [Grid.Unused.C4N; a];
0237      Grid.C4N(a,:) = zeros(size(a,1),2);
0238      Grid.N4N = NewNeumann;
0239   end
0240
0241   %Check Regularity
0242   nze = setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E);
0243   Marked = [];
0244   HangingNodes = abs(Grid.NOE-Grid.NOE');
0245
0246   if nnz(HangingNodes)
0247
0248       for j=1:size(nze,2)
0249           LocElement = Grid.N4E(nze(j),1:4);
0250           ElementEdges = [LocElement([1 2 3])',LocElement([2 3 1])'];
0251           HangingEdgesCounter = 0;
0252           for n=1:3
0253               if HangingNodes(ElementEdges(n,2),ElementEdges(n,1))
0254                  HangingNode = ...
0255                          HangingNodes(ElementEdges(n,2),ElementEdges(n,1));
0256                  if HangingNodes(ElementEdges(n,2),HangingNode) ...
0257                     || HangingNodes(HangingNode,ElementEdges(n,1))
0258                     Marked = [Marked; nze(j)];
0259                  else
0260                     HangingEdgesCounter = HangingEdgesCounter + 1;
0261                  end
0262               end
0263           end
0264           if HangingEdgesCounter>1
0265               Marked = [Marked; nze(j)];
0266           end
0267       end
0268       if ~isempty(Marked)
0269         Marked = unique(Marked);
0270         MarkedElements = zeros(size(Grid.N4E,1),2);
0271         MarkedElements(Marked,1) = 1;
```

```
0272         %call refineGrid to ensure grid condition
0273         [Grid,LogStruct2] = refineGrid(Grid,MarkedElements,Driver);
0274       else
0275         Grid.N4E = greenCompletion(Grid.N4E,Grid.NOE,Grid.Unused.N4E);
0276       end
0277   end
0278
0279  %Finalize logging
0280   if Options.Logging
0281      LogStruct.PostSize.C4N = size(Grid.C4N,1);
0282      LogStruct.PostSize.N4E = ...
0283                          size(setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E),2);
0284      LogStruct.PostSize.N4D = size(Grid.N4D,2);
0285      LogStruct.PostSize.N4N = size(Grid.N4N,2);
0286      LogStruct.ComputationTime = cputime - CPU_time;
0287   end
0288
0289   if Options.DisplayOutput
0290     %graphical representation
0291     show(Grid,zeros(size(Grid.C4N,1),1),'Coarsened grid',1)
0292     view(0,90);
0293   end
0294
0295   if Options.ShowTime
0296       ComputationTimeCoarsening = cputime - CPU_time
0297   end
0298
0299  %end of coarsenGrid
0300 %--------------------------------------------------------------------------
0301 function Grid = coarsenFather(FatherToCoarse,Grid)
0302 %coarsenFather: red-coarsens FatherToCoarse
0303
0304       Children = Grid.F4E(FatherToCoarse,4:7);
0305       EdgesOfFather = [Grid.F4E(FatherToCoarse,[1 2 3])' ...
0306                        Grid.F4E(FatherToCoarse,[2 3 1])'];
0307
0308       for j=1:3
0309           if Grid.NOE(EdgesOfFather(j,2),EdgesOfFather(j,1))==0 ...
0310              && Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2))==0
0311               %BoundaryNode
0312               Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = -1;
0313           elseif (Grid.NOE(EdgesOfFather(j,2),EdgesOfFather(j,1))==0)
0314               %HangingNode
0315               Grid.C4N(Grid.NOE(EdgesOfFather(j,1),...
0316                        EdgesOfFather(j,2)),:) = [0 0];
0317               Grid.Unused.C4N =  ...
0318         [Grid.Unused.C4N; Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2))];
0319               Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = 0;
0320           else
```

```
0321              Grid.NOE(EdgesOfFather(j,1),EdgesOfFather(j,2)) = 0;
0322          end
0323      end
0324
0325      Grid.N4E(Children(1),:) = [Grid.F4E(FatherToCoarse,1:3)...
0326                                 Grid.F4E(FatherToCoarse,8) 0];
0327      Grid.N4E(Children(2:4),:) = 0;
0328      Grid.Unused.N4E = [Grid.Unused.N4E; Children(2:4)'];
0329      Grid.F4E(FatherToCoarse,:) = 0;
0330      Grid.Unused.F4E = [Grid.Unused.F4E; FatherToCoarse];
0331
0332  %end of coarsenFather
0333 %--------------------------------------------------------------------------
0334 function N4E = greenCompletion(N4E,NOE,UnusedElements)
0335 %greenCompletion: green-refine all elements of N4E with a hanging node
0336
0337  nze = setdiff(1:size(N4E,1),UnusedElements);
0338  Nnze = length(nze);
0339  HangingNodes = abs(NOE-NOE');
0340  a = N4E(nze,1:3)';
0341  I1 = reshape(a,3*Nnze,1);
0342  a = N4E(nze,[2 3 1])';
0343  I2 = reshape(a,3*Nnze,1);
0344  a = repmat(1:Nnze,3,1);
0345  Val = reshape(a,3*Nnze,1);
0346  ElementsOnEdges = sparse(I1,I2,Val,size(NOE,1),size(NOE,2));
0347  [I1,I2] = find(HangingNodes);
0348  [I1,I2,ElementsToConsider] = ...
0349            find(sparse(I1,I2,1,size(NOE,1),size(NOE,2)).*ElementsOnEdges);
0350  ElementsToConsider = sort(nze(ElementsToConsider));
0351  %prelocation
0352  Counter = size(N4E,1);
0353  N4E = [N4E; zeros(nnz(HangingNodes)/2,5)];
0354
0355
0356  for j=1:nnz(ElementsToConsider)
0357      Element = ElementsToConsider(j);
0358      %EdgesOfElement = [N4E(nze(j),[1 2 3])' N4E(nze(j),[2 3 1])'];
0359      for k=1:3
0360          if HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k))
0361              HangingNode = ...
0362                      HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k));
0363              Counter = Counter + 1;
0364              LocElement = N4E(Element,:);
0365              N4E(Element,:) = ...
0366                     [HangingNode LocElement(rem(k+1,3)+1) LocElement(k)...
0367                                                  LocElement(4) Counter];
0368              N4E(Counter,:) = [HangingNode LocElement(rem(k,3)+1)...
0369                           LocElement(rem(k+1,3)+1) LocElement(4) Element];
```

```
0370
0371                break;
0372            end
0373
0374        end
0375    end
0376
0377    N4E = N4E(1:Counter,:);
0378
0379  %end of greenCompletion
0380 %-------------------------------------------------------------------------
0381 function [N4E,MarkedElements] = greenReverseCompletion(N4E,MarkedElements)
0382 %greenReverseCompletion: reverse green completion
0383
0384    GreenElements = find(N4E(:,5));
0385    for j=1:size(GreenElements,1)
0386
0387        GreenBrother = N4E(GreenElements(j),5);
0388
0389        %Stop when coming to lastborns
0390        if GreenBrother<GreenElements(j)
0391            N4E = N4E(1:GreenElements(j)-1,:);
0392            MarkedElements = MarkedElements(1:GreenElements(j)-1,:);
0393            return;
0394        end
0395
0396        %Reversing green splitting
0397        ThirdNode =  setdiff([N4E(GreenBrother,2:3)],...
0398                             [N4E(GreenElements(j),2:3)]);
0399        N4E(GreenElements(j),1:3) = [N4E(GreenElements(j),2)...
0400                                     N4E(GreenElements(j),3) ThirdNode];
0401        N4E(GreenElements(j),5) = 0;
0402
0403        %Update MarkedElements
0404        if MarkedElements(GreenBrother,1) %offensive refinement
0405              MarkedElements(GreenElements(j),1) = 1;
0406        end
0407
0408        if MarkedElements(GreenElements(j),2) ...%defensive coarsening
0409           && ~MarkedElements(GreenBrother,2)
0410            MarkedElements(GreenElements(j),2) = 0;
0411        end
0412
0413    end
0414
0415  %end of greenReverseCompletion
```

A.3 The Function *parabolicSolver*

```
0001 function [U,Grid,LogStruct]=...                                    %ok
0002                  parabolicSolver(OldGrid,Grid,Uold,Told,T,Driver,varargin)
0003 %   parabolicSolver: Solves weak form of an parabolic equation finding
0004 %                   FE U for Grid
0005 %
0006 %       d/dt(u)-div(D(x,t)u) +C*Nabla(u)+R*u= f(x,t)  u=u(x,t) scalar fct
0007 %                                    u(x,t) = u_d(t) on Dirichlet-edge
0008 %                     D(x,t)*Nabla(u)*n(x) = g(x,t) on Neumann-edge
0009 %
0010 %
0011 %   INPUT ARGUMENTS
0012 %
0013 %   Grid,OldGrid: structures with fields C4N,N4E,N4D,N4N,F4E,NOE,Unused,
0014 %                                        M,H,b,DoubleArea
0015 %
0016 %   [C4N: Coordinates;a nx2-matrix, where n is the number of vertices
0017 %       in the grid.
0018 %       Format of ith row: 1st coordinate of ith vertex/2nd coordinate of
0019 %       ith vertex
0020 %
0021 %   [N4E: Elements;nx5-matrix, where n is the number of triangles in
0022 %       the grid.
0023 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0024 %       triangle/3rd vertex of ith triangle/father(referring to F4E-Matrix)
0025 %       /green brother]
0026 %
0027 %   [N4D: Dirichlet nodes; nx2-matrix, where n is the number of
0028 %       dirichlet edges.
0029 %       Format of ith row: 1st vertex of ith dirichlet edge/2nd vertex of
0030 %       ith dirichlet edge]
0031 %
0032 %   [N4N: Neumann nodes; see Dirichlet]
0033 %
0034 %   [F4E: Fathers; nx8-matrix, where n is the number of fathers in
0035 %       the grid. A father is a triangle which is split in 4 congruent
0036 %       subtriangles(called children) given by bisecting all edges of the
0037 %       father. A father is not in the actual Element-matrix, since it is
0038 %       not an actual element. His children may, but don't have to be in
0039 %       this matrix, they too can be splitted and may be in the
0040 %       Fathers-matrix;
0041 %       Format of ith row: 1st vertex of ith father/2nd vertex of ith
0042 %       father/3rd vertex of ith father/middle child/2nd child/3rd child
0043 %       /4th child/father of father
0044 %       Remark: column 1-3 referring to C4N-matrix
0045 %               column 4-7 referring to N4E-matrix
0046 %               column 8 referring to F4E-matrix ]
0047 %
```

```
0048 %   [NON: Nodes on Edges; nxn-matrix, where n is number of vertices.
0049 %       With NON the algorithm administers vertices (normal and
0050 %       hanging ones). In ith row/jth column there is the number of the
0051 %       vertex on the edge between the ith and jth vertice(referring to
0052 %       C4N-matrix) if there is none or node lies on a boundary edge
0053 %       the entry is 0]
0054 %
0055 %   [Unused.C4N/N4E/F4E: Unused rows from these matrices;
0056 %       Format: vertical vector with row number entries
0057 %
0058 %   [M,H,b,DoubleArea: Saved mass-matrix,
0059 %       transport-matrix, rhs and double areas for Grid.]
0060 %
0061 %   [Driver: name of the file, where the problem data is located; string]
0062 %
0063 %   OUTPUT ARGUMENTS:
0064 %
0065 %   [U : result FE approximation for Grid. Vertical nx1 vector where the
0066 %       ith value corresponds to the ith coordinate in the Grid.C4N-matrix]
0067 %
0068 %   REMARK: parabolicSolver saves the structur CGrid(a common refinement of
0069 %       OldGrid and Grid) for later use to CGrid.mat
0070 %
0071 %   [LogStruct is a log-struct which contains information about the
0072 %       solution process.]
0073 %
0074 %   See also commonGrid,refineGrid,coarsenGrid
0075 %
0076 %   Copyright (c) 2007 Philipp Wissgott
0077 %                      Vienna University of Technology
0078
0079 %************************** OPTIONS ************************************
0080
0081   CPU_time = cputime;
0082   OptionFile = feval(Driver,'OptionFile');
0083   Options = feval(OptionFile,'LoadOptions','Solver',Driver,Told,T);
0084   ValD = Options.Value.D;
0085   ValC = Options.Value.C;
0086   ValR = Options.Value.R;
0087   ValF = Options.Value.F;
0088   Options.Logging = 0;
0089
0090
0091 %******************** TO LOG OR NOT TO LOG ******************************
0092
0093   if nargout==3
0094          Options.Logging = 1;
0095   end
0096
```

```
0097 %******************** GET COMMON GRID ***********************************
0098
0099   %Common grid of OldGrid and Grid
0100   if (nargin==7) && varargin{1}
0101       load CGrid
0102       if isequal(OldGrid.C4N,Grid.C4N) && isequal(OldGrid.N4E,Grid.N4E)
0103           GridsAreEqual = 1;
0104       else
0105           GridsAreEqual = 0;
0106       end
0107   else
0108      [CGrid,GridsAreEqual] = commonGrid(OldGrid,Grid,Driver);
0109   end
0110
0111 %*************************** CHECK INPUT ********************************
0112
0113   %Renaming for readability
0114   C4N = Grid.C4N;
0115   N4E = Grid.N4E;
0116   N4D = Grid.N4D;
0117   N4N = Grid.N4N;
0118
0119   C4Nold = OldGrid.C4N;
0120   N4Eold = OldGrid.N4E;
0121   clear OldGrid.C4N OldGrid.N4E OldGrid.NOE
0122
0123   %Adding fourth/fifth column if not existing
0124   if size(N4Eold,2)==3
0125       N4Eold = [N4Eold zeros(size(N4Eold,1),2)];
0126   end
0127   if size(N4E,2)==3
0128       N4E = [N4E; zeros(2,size(N4E,1))];
0129   end
0130   if size(N4Eold,2)==4
0131       N4Eold = [N4Eold zeros(size(N4Eold,1),1)];
0132   end
0133   if size(N4E,2)==4
0134       N4E = [N4E; zeros(1,size(N4E,1))];
0135   end
0136
0137   if isempty(N4E)
0138       error('Grid.N4E-matrix empty!');
0139   end
0140
0141   if isempty(C4N) || isempty(setdiff(1:size(C4N,1),Grid.Unused.C4N))
0142       error('Grid.C4N-matrix empty!');
0143   end
0144
0145   if isempty(N4D)
```

```
0146        error('Grid.N4D-matrix empty!');
0147    end
0148
0149    if isempty(N4Eold)
0150        error('OldGrid.N4E-matrix empty!');
0151    end
0152
0153    if isempty(C4Nold) ...
0154                    ||  isempty(setdiff(1:size(C4Nold,1),OldGrid.Unused.C4N))
0155        error('OldGrid.C4N-matrix empty!');
0156    end
0157
0158    if isempty(N4D)
0159        error('OldGrid.N4D-matrix empty!');
0160    end
0161
0162    %Check number of rows in Uold
0163    Uold = Uold(1:size(OldGrid.C4N,1));
0164
0165    %AssemblyIdentifier minimizes computation by indicating already stored
0166    %data
0167    AssemblyIdentifier = zeros(1,3);
0168    M = OldGrid.M;
0169    H = OldGrid.H;
0170    b = OldGrid.b;
0171    DoubleArea = OldGrid.DoubleArea;
0172    if GridsAreEqual
0173        if ~isempty(M)
0174            AssemblyIdentifier(1) = 1;
0175        end
0176        if (Options.Dependence.D<=0 || Options.Dependence.D==2) ...
0177            && (Options.Dependence.C<=0 || Options.Dependence.C==2) &&...
0178            (Options.Dependence.R<=0 || Options.Dependence.R==2)
0179            if ~isempty(H)
0180                AssemblyIdentifier(2) = 1;
0181            end
0182        end
0183        if Options.Dependence.F<=0 || Options.Dependence.F==2
0184            if ~isempty(b)
0185                AssemblyIdentifier(3) = 1;
0186            end
0187        end
0188    end
0189    if ~isempty(Grid.M)
0190        AssemblyIdentifier(1) = 1;
0191        M = Grid.M;
0192        DoubleArea = Grid.DoubleArea;
0193    end
0194    if ~isempty(Grid.H) ...
```

```
0195            && (Options.Dependence.D<=0 || Options.Dependence.D==2) ...
0196            && (Options.Dependence.C<=0 || Options.Dependence.C==2) &&...
0197            (Options.Dependence.R<=0 || Options.Dependence.R==2)
0198        AssemblyIdentifier(2) = 1;
0199        H = Grid.H;
0200    end
0201    if ~isempty(Grid.b)...
0202                        && (Options.Dependence.F<=0 || Options.Dependence.F==2)
0203        AssemblyIdentifier(3) = 1;
0204        b = Grid.b;
0205    end
0206
0207    clear OldGrid.M OldGrid.H OldGrid.b
0208
0209 %****************************** MAIN ************************************
0210
0211    %Initialisation
0212    UsedNodes = setdiff(1:size(C4N,1),Grid.Unused.C4N);
0213    BoundaryNodes = unique(N4D);
0214    FreeNodes = setdiff(UsedNodes,BoundaryNodes);
0215    if nnz(FreeNodes)>1e5
0216        Options.LargeDimensionMode = 1;
0217    else
0218        Options.LargeDimensionMode = 0;
0219    end
0220    nze = setdiff(1:size(N4E,1),Grid.Unused.N4E);
0221    U = zeros(size(C4N,1),1);
0222
0223    %Assembly
0224    %Prelocating vectors for assembly
0225    a = [N4E(nze,1) N4E(nze,1) N4E(nze,1) N4E(nze,2) N4E(nze,2)...
0226         N4E(nze,2) N4E(nze,3) N4E(nze,3) N4E(nze,3)]';
0227    I2 = reshape(a,9*nnz(nze),1);
0228    a = [N4E(nze,1:3) N4E(nze,1:3) N4E(nze,1:3)]';
0229    I1 = reshape(a,9*nnz(nze),1);
0230    LocMassMatrices = zeros(9*nnz(nze),1);
0231    Nnze = nnz(nze);
0232
0233    %the mass matrix for the reference element
0234    RefM = [2 1 1 1 2 1 1 1 2]/24;
0235
0236    if ~nnz(AssemblyIdentifier)
0237
0238       %Vertices
0239       A = C4N(N4E(nze,1),:);
0240       B = C4N(N4E(nze,2),:);
0241       C = C4N(N4E(nze,3),:);
0242       %Assembly of areas
0243       DoubleArea = B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
```

```
0244                    -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0245       %Assembly of mass matrix
0246       LocMassMatrices = reshape(RefM'*DoubleArea',9*Nnze,1);
0247       %Assembly of gradients
0248       N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)];
0249       N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)];
0250       N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)];
0251       %Assembly of transport matrix
0252       LocHMatrices = ...
0253    stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options) ...
0254    + convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options) ...
0255    + reactionMatrix(ValR,A,B,C,DoubleArea,LocMassMatrices,Told,T,Driver,...
0256                                                                  Options);
0257       %Assembly of RHS
0258       LocBs = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options);
0259
0260       M = sparse(I1,I2,LocMassMatrices,size(C4N,1),size(C4N,1));
0261       H = sparse(I1,I2,LocHMatrices,size(C4N,1),size(C4N,1));
0262       I1 = reshape(N4E(nze,1:3)',3*nnz(nze),1);
0263       b = sparse(I1,ones(nnz(I1),1),LocBs,size(C4N,1),1);
0264    end
0265
0266    if isequal(AssemblyIdentifier,[1 0 0])
0267
0268       %Vertices
0269       A = C4N(N4E(nze,1),:);
0270       B = C4N(N4E(nze,2),:);
0271       C = C4N(N4E(nze,3),:);
0272       %Assembly of gradients
0273       N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)];
0274       N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)];
0275       N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)];
0276       %Assembly of stiffness matrix
0277       LocHMatrices = ...
0278    stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options) ...
0279    + convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options) ...
0280    + reactionMatrix(ValR,A,B,C,DoubleArea,LocMassMatrices,Told,T,Driver,...
0281                                                                  Options);
0282       %Assembly of RHS
0283       LocBs = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options);
0284
0285       H = sparse(I1,I2,LocHMatrices,size(C4N,1),size(C4N,1));
0286        if ~LocMassMatrices
0287            if Options.Dependence.R<1
0288               H = H + ValR*M;
0289            elseif Options.Dependence.R==1
0290               R = Options.Theta*ValR(2) + (1-Options.Theta)*ValR(1);
0291               H = H + R*M;
0292            end
```

```
0293        end
0294      I1 = reshape(N4E(nze,1:3)',3*nnz(nze),1);
0295      b = sparse(I1,ones(size(I1,1),1),LocBs,size(C4N,1),1);
0296    end
0297
0298    if isequal(AssemblyIdentifier,[1 1 0])
0299
0300      %Vertices
0301      A = C4N(N4E(nze,1),:);
0302      B = C4N(N4E(nze,2),:);
0303      C = C4N(N4E(nze,3),:);
0304      %Assembly of RHS
0305      LocBs = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options);
0306
0307      I1 = reshape(N4E(nze,1:3)',3*nnz(nze),1);
0308      b = sparse(I1,ones(nnz(I1),1),LocBs,size(C4N,1),1);
0309    end
0310
0311    if isequal(AssemblyIdentifier,[1 0 1])
0312
0313      %Vertices
0314      A = C4N(N4E(nze,1),:);
0315      B = C4N(N4E(nze,2),:);
0316      C = C4N(N4E(nze,3),:);
0317      %Assembly of gradients
0318      N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)];
0319      N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)];
0320      N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)];
0321      %Assembly of stiffness matrix
0322      LocHMatrices = ...
0323    stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options) ...
0324    + convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options) ...
0325    + reactionMatrix(ValR,A,B,C,DoubleArea,LocMassMatrices,Told,T,Driver,...
0326                                                                Options);
0327      H = sparse(I1,I2,LocHMatrices,size(C4N,1),size(C4N,1));
0328      if ~LocMassMatrices
0329          H = H + ValR*M;
0330      end
0331    end
0332
0333    Grid.b = b;
0334    clear LocMassMatrices LocHMatrices LocBs a UsedNodes
0335
0336
0337    %Neumann
0338    for j = 1 : size(N4N,1)
0339
0340        Locg = (Options.Theta*feval(Driver,'g(x,t)',...
0341                                            sum(C4N(N4N(j,:),:))/2,T)...
```

```
0342              +(1-Options.Theta)*feval(Driver,'g(x,t)',...
0343                                                  sum(C4N(N4N(j,:),:))/2,Told));
0344      b(N4N(j,:))= b(N4N(j,:)) + norm(C4N(N4N(j,1),:) ...
0345                                                    - C4N(N4N(j,2),:))*Locg/2;
0346  end
0347
0348  %Tranfermatrix to old grid
0349  if GridsAreEqual
0350      Transma = M - (T-Told)*(1-Options.Theta)*H;
0351  else
0352
0353     %rows=new;columns=old
0354     [LocTransferMatrices,I1,I2] = transferMatrix(CGrid,C4N,N4E,...
0355           C4Nold,N4Eold,ValD,ValC,ValR,Told,T,Driver,Options);
0356     Transma =sparse(I1,I2,LocTransferMatrices,size(C4N,1),size(C4Nold,1));
0357  end
0358
0359  clear OldGrid N4E N4Eold N4N N4D;
0360
0361  %Boundary Condition
0362  U(BoundaryNodes) = feval(Driver,'u(x,t)',C4N(BoundaryNodes,:),T);
0363
0364  %RHS
0365  b = Transma*Uold + (T-Told) * b - (M + (T-Told) * Options.Theta * H)*U;
0366
0367  clear Transma LocTransferMatrices I1 I2 C4N C4Nold nze;
0368
0369  %prepare for solving
0370  H2 = M + (T-Told) * Options.Theta * H;
0371  H2 = H2(FreeNodes,FreeNodes);
0372  b2 = b(FreeNodes);
0373
0374  %Solving
0375  if Options.LargeDimensionMode
0376
0377      %clear workspace, save temporal workspace
0378      save temp;
0379      save temp2 H2 b2;
0380      clear all;
0381      load temp2
0382
0383      X = H2\b2;
0384
0385      %load temporal workspace
0386      load temp
0387      U(FreeNodes) = X;
0388
0389  else
0390      U(FreeNodes) = H2\b2;
```

```
0391    end
0392
0393 %*****************************VARIABLE OUTPUT*****************************
0394
0395    %store data
0396    Grid.M = M;
0397    Grid.H = H;
0398    Grid.DoubleArea = DoubleArea;
0399    if GridsAreEqual
0400        CGrid.M = M;
0401        CGrid.DoubleArea = DoubleArea;
0402    else
0403        CGrid.M = [];
0404    end
0405    save CGrid CGrid;
0406
0407    if Options.DisplayOutput
0408        % graphic representation
0409        Title = sprintf('Solution for %s at %.2f',Driver,T);
0410        show(Grid,U,Title,1)
0411    end
0412
0413    %Finalize logging
0414    if Options.Logging
0415        LogStruct.EnergyNorm = U'*(M + (T-Told) * Options.Theta * H)*U;
0416        LogStruct.ComputationTime = cputime - CPU_time;
0417    end
0418
0419    if Options.ShowTime
0420        ComputationTimeSolver = cputime - CPU_time
0421    end
0422
0423    %end of parabolicSolver
0424 %-------------------------------------------------------------------------
0425 function StiffnessMatrix= ...
0426        stiffnessMatrix(ValD,A,B,C,N1,N2,N3,DoubleArea,Told,T,Driver,Options)
0427 %computes stiffness matrix
0428
0429    switch (Options.Dependence.D)
0430        case 0
0431            ValD = ValD/2;
0432            N1Col = (ValD*N1')';
0433            N2Col = (ValD*N2')';
0434            N3Col = (ValD*N3')';
0435
0436        case 1
0437            ValD =...
0438                 (Options.Theta*ValD(3:4,:)+(1-Options.Theta)*ValD(1:2,:))/2;
0439            N1Col = (ValD*N1')';
```

```
0440            N2Col = (ValD*N2')';
0441            N3Col = (ValD*N3')';
0442
0443        case 2
0444            ValD = zeros(2*size(A,1),2);
0445            for j=1:size(Options.QWeights,1)
0446
0447                QuadraturVertices = ...
0448                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0449                ValD = ValD + Options.QWeights(j)*...
0450                                feval(Driver,'D(x,t)',QuadraturVertices,0);
0451            end
0452            ValD =reshape([ValD(1:size(A,1),1)';ValD(size(A,1)+1:end,1)';...
0453                           ValD(1:size(A,1),2)';ValD(size(A,1)+1:end,2)'],...
0454                                                              2,2*size(A,1))';
0455            ValD = blktridiag(ValD,size(A,1));
0456            N1Col = reshape(ValD*reshape(N1',2*size(A,1),1),2,size(A,1))';
0457            N2Col = reshape(ValD*reshape(N2',2*size(A,1),1),2,size(A,1))';
0458            N3Col = reshape(ValD*reshape(N3',2*size(A,1),1),2,size(A,1))';
0459
0460        case 3
0461            ValD = zeros(2*size(A,1),2);
0462
0463            for j=1:size(Options.QWeights,1)
0464
0465                QuadraturVertices = ...
0466                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0467                ValD = ValD + Options.Theta*Options.QWeights(j)*...
0468                                feval(Driver,'D(x,t)',QuadraturVertices,T);
0469                if Options.Theta~=1
0470                    ValD = ValD + (1-Options.Theta)*Options.QWeights(j)*...
0471                                feval(Driver,'D(x,t)',QuadraturVertices,Told);
0472                end
0473            end
0474            ValD =reshape([ValD(1:size(A,1),1)';ValD(size(A,1)+1:end,1)';...
0475                           ValD(1:size(A,1),2)';ValD(size(A,1)+1:end,2)'],...
0476                                                              2,2*size(A,1))';
0477            ValD = blktridiag(ValD,size(A,1));
0478            N1Col = reshape(ValD*reshape(N1',2*size(A,1),1),2,size(A,1))';
0479            N2Col = reshape(ValD*reshape(N2',2*size(A,1),1),2,size(A,1))';
0480            N3Col = reshape(ValD*reshape(N3',2*size(A,1),1),2,size(A,1))';
0481    end
0482
0483    a = [(N1(:,1).*N1Col(:,1) + N1(:,2).*N1Col(:,2))./DoubleArea,...
0484         (N2(:,1).*N1Col(:,1) + N2(:,2).*N1Col(:,2))./DoubleArea,...
0485         (N3(:,1).*N1Col(:,1) + N3(:,2).*N1Col(:,2))./DoubleArea,...
0486         (N1(:,1).*N2Col(:,1) + N1(:,2).*N2Col(:,2))./DoubleArea,...
0487         (N2(:,1).*N2Col(:,1) + N2(:,2).*N2Col(:,2))./DoubleArea,...
0488         (N3(:,1).*N2Col(:,1) + N3(:,2).*N2Col(:,2))./DoubleArea,...
```

```
0489        (N1(:,1).*N3Col(:,1) + N1(:,2).*N3Col(:,2))./DoubleArea,...
0490        (N2(:,1).*N3Col(:,1) + N2(:,2).*N3Col(:,2))./DoubleArea,...
0491        (N3(:,1).*N3Col(:,1) + N3(:,2).*N3Col(:,2))./DoubleArea];
0492   StiffnessMatrix = reshape(a',9*size(N1,1),1);
0493
0494   %end of stiffnessMatrix
0495 %-------------------------------------------------------------------------
0496 function ConvectionMatrix = ...
0497                 convectionMatrix(ValC,A,B,C,N1,N2,N3,Told,T,Driver,Options)
0498 %computes convection matrix
0499
0500    ConvectionMatrix = zeros(9*size(N1,1),1);
0501    switch (Options.Dependence.C)
0502        case 0
0503             a = [N1*ValC',N1*ValC',N1*ValC',...
0504                  N2*ValC',N2*ValC',N2*ValC',...
0505                  N3*ValC',N3*ValC',N3*ValC']/6;
0506            ConvectionMatrix = reshape(a',9*size(N1,1),1);
0507        case 1
0508           ValC = Options.Theta*ValC(1,:) + (1-Options.Theta)*ValC(2,:);
0509           a = [N1*ValC',N1*ValC',N1*ValC',...
0510                N2*ValC',N2*ValC',N2*ValC',...
0511                N3*ValC',N3*ValC',N3*ValC']/6;
0512            ConvectionMatrix = reshape(a',9*size(N1,1),1);
0513
0514        case 2
0515           for j=1:size(Options.QWeights,1)
0516
0517               QuadraturVertices = ...
0518                     A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0519               ValC = feval(Driver,'c(x,t)',QuadraturVertices,0);
0520               a = [dot(N1,ValC,2)*Options.QBasisfct(j,1),...
0521                    dot(N1,ValC,2)*Options.QBasisfct(j,2),...
0522                    dot(N1,ValC,2)*Options.QBasisfct(j,3),...
0523                    dot(N2,ValC,2)*Options.QBasisfct(j,1),...
0524                    dot(N2,ValC,2)*Options.QBasisfct(j,2),...
0525                    dot(N2,ValC,2)*Options.QBasisfct(j,3),...
0526                    dot(N3,ValC,2)*Options.QBasisfct(j,1),...
0527                    dot(N3,ValC,2)*Options.QBasisfct(j,2),...
0528                    dot(N3,ValC,2)*Options.QBasisfct(j,3)];
0529               ConvectionMatrix = ConvectionMatrix + ...
0530                           Options.QWeights(j)*reshape(a',9*size(N1,1),1);
0531           end
0532        case 3
0533           for j=1:size(Options.QWeights,1)
0534
0535               QuadraturVertices = ...
0536                     A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0537               ValC = Options.Theta*...
```

```
0538                                feval(Driver,'c(x,t)',QuadraturVertices,T);
0539                if Options.Theta~=1
0540                    ValC = ValC + (1-Options.Theta)*...
0541                              feval(Driver,'c(x,t)',QuadraturVertices,Told);
0542                end
0543                a = [dot(N1,ValC,2)*Options.QBasisfct(j,1),...
0544                     dot(N1,ValC,2)*Options.QBasisfct(j,2),...
0545                     dot(N1,ValC,2)*Options.QBasisfct(j,3),...
0546                     dot(N2,ValC,2)*Options.QBasisfct(j,1),...
0547                     dot(N2,ValC,2)*Options.QBasisfct(j,2),...
0548                     dot(N2,ValC,2)*Options.QBasisfct(j,3),...
0549                     dot(N3,ValC,2)*Options.QBasisfct(j,1),...
0550                     dot(N3,ValC,2)*Options.QBasisfct(j,2),...
0551                     dot(N3,ValC,2)*Options.QBasisfct(j,3)];
0552                ConvectionMatrix = ConvectionMatrix + ...
0553                              Options.QWeights(j)*reshape(a',9*size(N1,1),1);
0554            end
0555
0556     end
0557
0558   %end of convectionMatrix
0559 %-------------------------------------------------------------------------
0560 function ReactionMatrix = ...
0561      reactionMatrix(ValR,A,B,C,DoubleArea,MassMatrix,Told,T,Driver,Options)
0562 %computes reaction matrix
0563
0564     switch (Options.Dependence.R)
0565
0566       case -1
0567           ReactionMatrix = zeros(9*size(A,1),1);
0568       case 0
0569           ReactionMatrix = ValR*MassMatrix;
0570       case 1
0571           ValR = Options.Theta*ValR(2) + (1-Options.Theta)*ValR(1);
0572           ReactionMatrix = ValR*MassMatrix;
0573
0574       case 2
0575           ReactionMatrix = zeros(9*size(A,1),1);
0576           for j=1:size(Options.QWeights,1)
0577               BasisPairs = Options.QBasisfct(j,:)'*Options.QBasisfct(j,:);
0578               QuadraturVertices = ...
0579                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0580               ValR = feval(Driver,'r(x,t)',QuadraturVertices,0);
0581               ValR = reshape(repmat(ValR.*DoubleArea,1,9)',9*size(A,1),1);
0582               BasisPairs = repmat(reshape(BasisPairs,9,1),size(A,1),1);
0583               ReactionMatrix = ReactionMatrix + ...
0584                                    Options.QWeights(j)*ValR.*BasisPairs;
0585           end
0586       case 3
```

```
0587            ReactionMatrix = zeros(9*size(A,1),1);
0588            for j=1:size(Options.QWeights,1)
0589                BasisPairs = Options.QBasisfct(j,:)'*Options.QBasisfct(j,:);
0590                QuadraturVertices = ...
0591                       A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0592                ValR = Options.Theta*...
0593                                  feval(Driver,'r(x,t)',QuadraturVertices,T);
0594                if Options.Theta~=1
0595                    ValR = ValR + (1-Options.Theta)*...
0596                                feval(Driver,'r(x,t)',QuadraturVertices,Told);
0597                end
0598                ValR = reshape(repmat(ValR.*DoubleArea,1,9)',9*size(A,1),1);
0599                BasisPairs = repmat(reshape(BasisPairs,9,1),size(A,1),1);
0600                ReactionMatrix = ReactionMatrix + ...
0601                                        Options.QWeights(j)*ValR.*BasisPairs;
0602            end
0603        end
0604
0605      %end of reactionMatrix
0606    %-------------------------------------------------------------------------
0607    function RHS = rightHandSide(ValF,A,B,C,DoubleArea,Told,T,Driver,Options)
0608    %right hand side
0609
0610       switch (Options.Dependence.F)
0611            case -1
0612                RHS = zeros(3*size(A,1),1);
0613            case 0
0614                RHS = 1/6 * ValF * ones(3*size(A,1),1);
0615            case 1
0616                RHS = 1/6 * (Options.Theta*ValF(2)+...
0617                             (1-Options.Theta)*ValF(1)) * ones(3*size(A,1),1);
0618            case 2
0619                a = zeros(3,size(A,1));
0620                for j =1:size(Options.QWeights,1)
0621                    a = a + Options.QWeights(j)*Options.QBasisfct(j,:)'...
0622                     *(feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0623                                (C-A)*Options.QNodes(j,2),0).*DoubleArea)';
0624                end
0625                RHS = reshape(a,3*size(A,1),1);
0626            case 3
0627                a = zeros(3,size(A,1));
0628                for j =1:size(Options.QWeights,1)
0629                    a = a + ...
0630                    Options.Theta*Options.QWeights(j)*Options.QBasisfct(j,:)'...
0631                     *(feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0632                                (C-A)*Options.QNodes(j,2),T).*DoubleArea)';
0633                    if Options.Theta~=1
0634                        a = a + (1-Options.Theta)*...
0635                                 Options.QWeights(j)*Options.QBasisfct(j,:)'...
```

```
0636                     *(feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0637                             (C-A)*Options.QNodes(j,2),Told).*DoubleArea)';
0638                 end
0639             end
0640             RHS = reshape(a,3*size(A,1),1);
0641
0642     end
0643
0644   %end of locB
0645 %-------------------------------------------------------------------------
0646 function [TransferMatrix,I1,I2] = transferMatrix(CGrid,C4N,N4E,...
0647                   C4Nold,N4Eold,ValD,ValC,ValR,Told,T,Driver,Options)
0648 %computes transfer matrix by transforming to
0649 %reference element
0650
0651     nze = setdiff(1:size(CGrid.N4E,1),CGrid.Unused.N4E);
0652     A = CGrid.C4N(CGrid.N4E(nze,1),:);
0653     B = CGrid.C4N(CGrid.N4E(nze,2),:);
0654     C = CGrid.C4N(CGrid.N4E(nze,3),:);
0655     CGrid.DoubleArea = B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
0656                         -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0657     Nodes = N4E(CGrid.N4E(nze,4),1:3);
0658     OldNodes = N4Eold(CGrid.N4E(nze,5),1:3);
0659     NewA = C4N(Nodes(:,1),:);
0660     NewB = C4N(Nodes(:,2),:);
0661     NewC = C4N(Nodes(:,3),:);
0662     OldA = C4Nold(OldNodes(:,1),:);
0663     OldB = C4Nold(OldNodes(:,2),:);
0664     OldC = C4Nold(OldNodes(:,3),:);
0665     I1 = reshape([Nodes Nodes Nodes]',9*size(Nodes,1),1);
0666     I2 = reshape(OldNodes(:,[1 1 1 2 2 2 3 3 3])',9*size(OldNodes,1),1);
0667
0668     MassMatrix = zeros(9*size(A,1),1);
0669     ConvectionMatrix = zeros(9*size(A,1),1);
0670     ReactionMatrix = zeros(9*size(A,1),1);
0671
0672     for j =1:size(Options.QWeights,1)
0673
0674         PsiQNodes = A + (B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0675         M1 = reshape([NewB-NewA NewC-NewA]',2,2*size(NewA,1))';
0676         M1 = blktridiag(M1,size(NewA,1));
0677         QNodesNew = M1\reshape((PsiQNodes - NewA)',2*size(NewA,1),1);
0678         QNodesNew = reshape(QNodesNew,2,size(NewA,1))';
0679         BasisPhiNew =[ones(size(NewA,1),1)-QNodesNew(:,1)-QNodesNew(:,2) ...
0680                                                 QNodesNew(:,1) QNodesNew(:,2)];
0681         M2 = reshape([OldB-OldA OldC-OldA]',2,2*size(OldA,1))';
0682         M2 = blktridiag(M2,size(OldA,1));
0683         QNodesOld = M2\reshape((PsiQNodes - OldA)',2*size(OldA,1),1);
0684         QNodesOld = reshape(QNodesOld,2,size(OldA,1))';
```

```
0685        BasisPhiOld =[ones(size(NewA,1),1)-QNodesOld(:,1)-QNodesOld(:,2) ...
0686                                              QNodesOld(:,1) QNodesOld(:,2)];
0687        a = [BasisPhiNew(:,1).*BasisPhiOld(:,1).*CGrid.DoubleArea,...
0688             BasisPhiNew(:,2).*BasisPhiOld(:,1).*CGrid.DoubleArea,...
0689             BasisPhiNew(:,3).*BasisPhiOld(:,1).*CGrid.DoubleArea,...
0690             BasisPhiNew(:,1).*BasisPhiOld(:,2).*CGrid.DoubleArea,...
0691             BasisPhiNew(:,2).*BasisPhiOld(:,2).*CGrid.DoubleArea,...
0692             BasisPhiNew(:,3).*BasisPhiOld(:,2).*CGrid.DoubleArea,...
0693             BasisPhiNew(:,1).*BasisPhiOld(:,3).*CGrid.DoubleArea,...
0694             BasisPhiNew(:,2).*BasisPhiOld(:,3).*CGrid.DoubleArea,...
0695             BasisPhiNew(:,3).*BasisPhiOld(:,3).*CGrid.DoubleArea];
0696        MassMatrix = MassMatrix +...
0697                             Options.QWeights(j)* reshape(a',9*size(A,1),1);
0698
0699        if Options.Theta~=1
0700
0701             DoubleArea = ...
0702      NewB(:,1).*NewC(:,2) + NewC(:,1).*NewA(:,2) + NewA(:,1).*NewB(:,2) ...
0703    - NewB(:,1).*NewA(:,2) - NewC(:,1).*NewB(:,2) - NewA(:,1).*NewC(:,2) ;
0704             OldDoubleArea = ...
0705      OldB(:,1).*OldC(:,2) + OldC(:,1).*OldA(:,2) + OldA(:,1).*OldB(:,2) ...
0706    - OldB(:,1).*OldA(:,2) - OldC(:,1).*OldB(:,2) - OldA(:,1).*OldC(:,2) ;
0707             N1New = [NewB(:,2)-NewC(:,2) NewC(:,1)-NewB(:,1)]./...
0708                                                  (DoubleArea*ones(1,2));
0709             N2New = [NewC(:,2)-NewA(:,2) NewA(:,1)-NewC(:,1)]./...
0710                                                  (DoubleArea*ones(1,2));
0711             N3New = [NewA(:,2)-NewB(:,2) NewB(:,1)-NewA(:,1)]./...
0712                                                  (DoubleArea*ones(1,2));
0713             N1Old = [OldB(:,2)-OldC(:,2) OldC(:,1)-OldB(:,1)]./...
0714                                                (OldDoubleArea*ones(1,2));
0715             N2Old = [OldC(:,2)-OldA(:,2) OldA(:,1)-OldC(:,1)]./...
0716                                                (OldDoubleArea*ones(1,2));
0717             N3Old = [OldA(:,2)-OldB(:,2) OldB(:,1)-OldA(:,1)]./...
0718                                                (OldDoubleArea*ones(1,2));
0719
0720             %Assembly of convection matrix
0721             switch (Options.Dependence.C)
0722                 case -1
0723                    CValues = zeros(size(N1New,1),2);
0724                 case 0
0725                    CValues = repmat(ValC,size(N1New,1),1);
0726                 case 1
0727                    CValues = Options.Theta*ValC(1,:) + ...
0728                                             (1-Options.Theta)*ValC(2,:);
0729                    CValues = repmat(CValues,size(N1New,1),1);
0730                 case 2
0731                    CValues = feval(Driver,'c(x,t)',PsiQNodes,0);
0732                  case 3
0733                    CValues = ...
```

```
0734                        Options.Theta*feval(Driver,'c(x,t)',PsiQNodes,T);
0735                  if Options.Theta~=1
0736                       CValues = CValues + (1-Options.Theta)*...
0737                                  feval(Driver,'c(x,t)',PsiQNodes,Told);
0738                  end
0739             end
0740             a = [dot(N1Old,CValues,2).*BasisPhiNew(:,1),...
0741                  dot(N1Old,CValues,2).*BasisPhiNew(:,2),...
0742                  dot(N1Old,CValues,2).*BasisPhiNew(:,3),...
0743                  dot(N2Old,CValues,2).*BasisPhiNew(:,1),...
0744                  dot(N2Old,CValues,2).*BasisPhiNew(:,2),...
0745                  dot(N2Old,CValues,2).*BasisPhiNew(:,3),...
0746                  dot(N3Old,CValues,2).*BasisPhiNew(:,1),...
0747                  dot(N3Old,CValues,2).*BasisPhiNew(:,2),...
0748                  dot(N3Old,CValues,2).*BasisPhiNew(:,3)];
0749             ConvectionMatrix = ConvectionMatrix +...
0750                Options.QWeights(j)*reshape(a',9*size(A,1),1).*...
0751                      reshape(repmat(CGrid.DoubleArea,1,9)',9*size(A,1),1);
0752
0753             %Assembly of reaction matrix
0754             BasisPairs = [BasisPhiNew(:,1).*BasisPhiOld(:,1),...
0755                           BasisPhiNew(:,2).*BasisPhiOld(:,1),...
0756                           BasisPhiNew(:,3).*BasisPhiOld(:,1),...
0757                           BasisPhiNew(:,1).*BasisPhiOld(:,2),...
0758                           BasisPhiNew(:,2).*BasisPhiOld(:,2),...
0759                           BasisPhiNew(:,3).*BasisPhiOld(:,2),...
0760                           BasisPhiNew(:,1).*BasisPhiOld(:,3),...
0761                           BasisPhiNew(:,2).*BasisPhiOld(:,3),...
0762                           BasisPhiNew(:,3).*BasisPhiOld(:,3)];
0763             BasisPairs = reshape(BasisPairs',9*size(A,1),1);
0764             RValues = Options.Theta*feval(Driver,'r(x,t)',PsiQNodes,T);
0765             if Options.Theta~=1
0766                  RValues = RValues + (1-Options.Theta)*...
0767                                  feval(Driver,'r(x,t)',PsiQNodes,Told);
0768             end
0769             RValues = reshape(repmat(RValues.*CGrid.DoubleArea,1,9)',...
0770                                                          9*size(A,1),1);
0771             ReactionMatrix = ReactionMatrix + ...
0772                                  Options.QWeights(j)*RValues.*BasisPairs;
0773         end
0774
0775     end
0776
0777     if Options.Theta~=1
0778
0779           %Assembly of stiffnessmatrix
0780           switch (Options.Dependence.D)
0781                case 0
0782                    ValD = ValD/2;
```

```
0783                    N1Col = (ValD*N1Old')';
0784                    N2Col = (ValD*N2Old')';
0785                    N3Col = (ValD*N3Old')';
0786
0787                case 1
0788                    ValD = (Options.Theta*ValD(3:4,:)+...
0789                                        (1-Options.Theta)*ValD(1:2,:))/2;
0790                    N1Col = (ValD*N1Old')';
0791                    N2Col = (ValD*N2Old')';
0792                    N3Col = (ValD*N3Old')';
0793
0794                case 2
0795                    ValD = zeros(2*size(A,1),2);
0796                    for j=1:size(Options.QWeights,1)
0797
0798                         QuadraturVertices = ...
0799                                             A+(B-A)*Options.QNodes(j,1)+...
0800                                            (C-A)*Options.QNodes(j,2);
0801                         ValD = ValD + Options.QWeights(j)*...
0802                                 feval(Driver,'D(x,t)',QuadraturVertices,0);
0803                    end
0804                    ValD = reshape([ValD(1:size(A,1),1)';...
0805                                    ValD(size(A,1)+1:end,1)';...
0806                                    ValD(1:size(A,1),2)';...
0807                                    ValD(size(A,1)+1:end,2)'],...
0808                                                        2,2*size(A,1))';
0809                    ValD = blktridiag(ValD,size(A,1));
0810                    N1Col = reshape(ValD*...
0811                                reshape(N1Old',2*size(A,1),1),2,size(A,1))';
0812                    N2Col = reshape(ValD*...
0813                                reshape(N2Old',2*size(A,1),1),2,size(A,1))';
0814                    N3Col = reshape(ValD*...
0815                                reshape(N3Old',2*size(A,1),1),2,size(A,1))';
0816
0817                case 3
0818                    ValD = zeros(2*size(A,1),2);
0819
0820                    for j=1:size(Options.QWeights,1)
0821
0822                        QuadraturVertices = ...
0823                                             A+(B-A)*Options.QNodes(j,1)+...
0824                                             (C-A)*Options.QNodes(j,2);
0825                        ValD = ValD+Options.Theta*Options.QWeights(j)*...
0826                                 feval(Driver,'D(x,t)',QuadraturVertices,T);
0827                        if Options.Theta~=1
0828                             ValD = ValD + (1-Options.Theta)*...
0829                                              Options.QWeights(j)*...
0830                              feval(Driver,'D(x,t)',QuadraturVertices,Told);
0831                        end
```

```
0832                    end
0833                    ValD = reshape([ValD(1:size(A,1),1)';...
0834                                    ValD(size(A,1)+1:end,1)';...
0835                                    ValD(1:size(A,1),2)';...
0836                                    ValD(size(A,1)+1:end,2)'],...
0837                                                        2,2*size(A,1))';
0838                    ValD = blktridiag(ValD,size(A,1));
0839                    N1Col = ...
0840                   reshape(ValD*reshape(N1Old',2*size(A,1),1),2,size(A,1))';
0841                    N2Col = ...
0842                   reshape(ValD*reshape(N2Old',2*size(A,1),1),2,size(A,1))';
0843                    N3Col = ...
0844                   reshape(ValD*reshape(N3Old',2*size(A,1),1),2,size(A,1))';
0845            end
0846
0847            a = [(N1New(:,1).*N1Col(:,1) + N1New(:,2).*N1Col(:,2)).*...
0848                                                        CGrid.DoubleArea,...
0849                 (N2New(:,1).*N1Col(:,1) + N2New(:,2).*N1Col(:,2)).*...
0850                                                        CGrid.DoubleArea,...
0851                 (N3New(:,1).*N1Col(:,1) + N3New(:,2).*N1Col(:,2)).*...
0852                                                        CGrid.DoubleArea,...
0853                 (N1New(:,1).*N2Col(:,1) + N1New(:,2).*N2Col(:,2)).*...
0854                                                        CGrid.DoubleArea,...
0855                 (N2New(:,1).*N2Col(:,1) + N2New(:,2).*N2Col(:,2)).*...
0856                                                        CGrid.DoubleArea,...
0857                 (N3New(:,1).*N2Col(:,1) + N3New(:,2).*N2Col(:,2)).*...
0858                                                        CGrid.DoubleArea,...
0859                 (N1New(:,1).*N3Col(:,1) + N1New(:,2).*N3Col(:,2)).*...
0860                                                        CGrid.DoubleArea,...
0861                 (N2New(:,1).*N3Col(:,1) + N2New(:,2).*N3Col(:,2)).*...
0862                                                        CGrid.DoubleArea,...
0863                 (N3New(:,1).*N3Col(:,1) + N3New(:,2).*N3Col(:,2)).*...
0864                                                          CGrid.DoubleArea];
0865            StiffnessMatrix = reshape(a',9*size(N1New,1),1);
0866
0867            TransferMatrix = MassMatrix - (T-Told)*(1-Options.Theta)*...
0868                           (StiffnessMatrix+ConvectionMatrix+ReactionMatrix);
0869     else
0870            TransferMatrix = MassMatrix;
0871     end
0872
0873
0874
0875   %end of transferMatrix
0876 %-------------------------------------------------------------------------
0877 function A = blktridiag(A,n)
0878 % BLKTRIDIAG: computes a sparse (block) tridiagonal matrix with n blocks
0879
0880     [q,p] = size(A);
```

```
0881      [ind1,ind2,ind3]=ndgrid(0:p-1,0:p-1,0:n-1);
0882      rind = 1+ind1(:)+p*ind3(:);
0883      cind = 1+ind2(:)+p*ind3(:);
0884      v = reshape(A',n*4,1);
0885      A = sparse(rind,cind,v,n*p,n*p);
0886
0887      %end of blktridiag
```

A.4 The Function *etaSpatial*

```
0001 function [MarkedElements,EtaSpatial,RefinementIndicator,LogStruct] =... %ok
0002                        etaSpatial(OldGrid,Grid,Uold,U,Told,T,Driver,Options)
0003 %markElements: Markes elements to be refined and coarsened
0004 %and computes error indicators
0005 %
0006 %   INPUT ARGUMENTS
0007 %
0008 %   Grid,OldGrid: structures with fields C4N,N4E,N4D,N4N,F4E,NOE,Unused,
0009 %                                       M,H,b,DoubleArea
0010 %
0011 %   [C4N: Coordinates;a nx2-matrix, where n is the number of vertices
0012 %       in the grid.
0013 %       Format of ith row: 1st coordinate of ith vertex/2nd coordinate of
0014 %       ith vertex
0015 %
0016 %   [N4E: Elements;nx5-matrix, where n is the number of triangles in
0017 %       the grid.F
0018 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0019 %       triangle/3rd vertex of ith triangle/father(referring to F4E-Matrix)
0020 %       /green brother]
0021 %
0022 %   [N4D: Dirichlet nodes; nx2-matrix, where n is the number of
0023 %       dirichlet edges.
0024 %       Format of ith row: 1st vertex of ith dirichlet edge/2nd vertex of
0025 %       ith dirichlet edge]
0026 %
0027 %   [N4N: Neumann nodes; see Dirichlet]
0028 %
0029 %   [F4E: Unused]
0030 %
0031 %   [NON: unused]
0032 %
0033 %   [Unused.C4N/N4E/F4E: Unused rows from these matrices;
0034 %       Format: vertical vector with row number entries
0035 %
0036 %   [M,H,b,DoubleArea: Saved mass-matrix,
0037 %       transport-matrix, rhs and double areas for Grid. ]
0038 %
0039 %   [Uold,U: the approximations for the coordinates of OldGrid and Grid,
```

```
0040 %       resp.; nx1 resp. mx1 if n is the number of vertices of OldGrid and
0041 %       n is the number of vertices of Grid]
0042 %
0043 %   [Told,T: time for Uold and U; scalar]
0044 %
0045 %   [Driver: name of the file where the problem data is located; string]
0046 %
0047 %   [Options: User set options, structure]
0048 %
0049 %   OUTPUT ARGUMENTS:
0050 %
0051 %   [MarkedElements: nx2 matrix, where n is the number of elements of Grid
0052 %       1 means marked, 0 otherwise]
0053 %
0054 %   [Eta: error estimator,structure with fields:
0055 %       SpatialErrorEstimator,VerfuerthErrorEstimator,DataErrorEstimator,
0056 %       FDataErrorEstiator,NeumannErrorEstimator]
0057 %
0058 %   [RefinementIndicator: The values that lead to MarkedElements; nx1 where
0059 %       n is the number of elements of Grid]
0060 %
0061 %   [LogStruct is a log-struct which contains information about the
0062 %      solution process.]
0063 %
0064 %   REMARK: expects a file CGrid.mat to be in the same directory storing
0065 %      a common grid of OldGrid and Grid
0066 %
0067 %   See also parabolicSolver,commonGrid
0068 %
0069 %   Copyright (c) 2007 Philipp Wissgott
0070 %                      Vienna University of Technology
0071 %
0072 %************************** LOGGING ************************************
0073
0074   CPU_time = cputime;
0075   if nargout==4
0076
0077       Options.Logging = 1;
0078   else
0079       Options.Logging = 0;
0080   end
0081
0082 %************************** OPTIONS ************************************
0083
0084   OptionFile = feval(Driver,'OptionFile');
0085   Options = feval(OptionFile,'LoadOptions','Indicator',Driver,Told,T);
0086   ValD = Options.Value.D;
0087   ValC = Options.Value.C;
0088   ValR = Options.Value.R;
```

```
0089   ValF = Options.Value.F;
0090   ValG = Options.Value.G;
0091   Lambda = Options.Lambda;  %parameter for energy norm
0092   Beta = Options.Beta; %parameter for energy norm
0093   Theta = Options.Theta;  %parameter for time discretization
0094
0095 %************************** MAIN ***************************************
0096
0097   %Initialization
0098   try
0099       load CGrid
0100       CommonGrid = CGrid;
0101   catch
0102       CommonGrid = commonGrid(OldGrid,Grid,Driver);
0103   end
0104   nze = find(CommonGrid.N4E(:,1));
0105   Nnze =nnz(nze);
0106   Rotation = [0 1;-1 0];
0107   if Theta==1
0108       BackwardValue = false;
0109   else
0110       BackwardValue = true;
0111   end
0112
0113   %Rename quadratur data for performance
0114   QWeights = Options.QWeights;
0115   QBasisfct = Options.QBasisfct;
0116   QNodes = Options.QNodes;
0117   NumberOfQWeights = size(QWeights,1);
0118   NumberOfQWeightsOnes = ones(NumberOfQWeights,1);
0119
0120   %Obtain values U,Uold at the vertices of CommonGrid
0121   UCommon = getCommonU(OldGrid,Grid,CommonGrid,Uold,U);
0122   if size(UCommon,1)>1e5
0123       Options.LargeDimensionMode = 1;
0124   else
0125       Options.LargeDimensionMode = 0;
0126   end
0127
0128   %Obtain L2 projections
0129   [Projection,DoubleArea] =l2Projections(CommonGrid,Told,T,Driver,Options);
0130   CommonGrid.DoubleArea = DoubleArea;
0131
0132   %prelocate variables
0133   ElementResiduals = zeros(Nnze,1);
0134   DataResiduals = zeros(Nnze,1);
0135   FDataErrorEstimators = zeros(Nnze,1);
0136   GContribution = sparse(size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0137
```

```
0138  %Prelocating vectors for assembly
0139  a = CommonGrid.N4E(nze,[2 3 1])';
0140  I2 = reshape(a,3*Nnze,1);
0141  a = CommonGrid.N4E(nze,1:3)';
0142  I1 = reshape(a,3*Nnze,1);
0143  LocTau = T-Told; %local time difference
0144
0145  %Collect nodes
0146  Nodes = CommonGrid.N4E(nze,1:3);
0147
0148  %Obtain U at vertices
0149  U1 = UCommon(Nodes(:,1),2);
0150  U2 = UCommon(Nodes(:,2),2);
0151  U3 = UCommon(Nodes(:,3),2);
0152  U1old = UCommon(Nodes(:,1),1);
0153  U2old = UCommon(Nodes(:,2),1);
0154  U3old = UCommon(Nodes(:,3),1);
0155
0156  %Obtain UTheta at vertices
0157  UTheta1 = Theta*U1+(1-Theta)*U1old;
0158  UTheta2 = Theta*U2+(1-Theta)*U2old;
0159  UTheta3 = Theta*U3+(1-Theta)*U3old;
0160
0161  %Collect vertices
0162  A = CommonGrid.C4N(CommonGrid.N4E(nze,1),:);
0163  B = CommonGrid.C4N(CommonGrid.N4E(nze,2),:);
0164  C = CommonGrid.C4N(CommonGrid.N4E(nze,3),:);
0165
0166  %Assembly of gradients
0167  N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)]./(DoubleArea*ones(1,2));
0168  N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)]./(DoubleArea*ones(1,2));
0169  N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)]./(DoubleArea*ones(1,2));
0170
0171  NablaUTheta = N1.*(U1*ones(1,2)) +...
0172                N2.*(U2*ones(1,2)) + ...
0173                N3.*(U3*ones(1,2));
0174
0175  %Collecting divDh values
0176  if Options.Dependence.D<2
0177      DivDhTerm = zeros(Nnze,1);
0178  else
0179      D1=Projection.D(Nodes(:,1),:,:);
0180      D2=Projection.D(Nodes(:,2),:,:);
0181      D3=Projection.D(Nodes(:,3),:,:);
0182      D1 = reshape(permute(D1,[2 1 3]),2*size(A,1),2);
0183      D2 = reshape(permute(D2,[2 1 3]),2*size(A,1),2);
0184      D3 = reshape(permute(D3,[2 1 3]),2*size(A,1),2);
0185      D1 = blktridiag(D1,size(A,1));
0186      D2 = blktridiag(D2,size(A,1));
```

```
0187        D3 = blktridiag(D3,size(A,1));
0188        D1NabU = ...
0189            reshape(D1*reshape(NablaUTheta',2*size(A,1),1),2,size(A,1))';
0190        D2NabU = ...
0191            reshape(D2*reshape(NablaUTheta',2*size(A,1),1),2,size(A,1))';
0192        D3NabU = ...
0193            reshape(D3*reshape(NablaUTheta',2*size(A,1),1),2,size(A,1))';
0194        a = D1NabU.*N1 + D2NabU.*N2 + D3NabU.*N3;
0195        DivDhTerm = a(:,1)+a(:,2);
0196    end
0197
0198    for j=1:size(QWeights,1)
0199
0200         QuadraturVertices = ...
0201                      A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0202
0203         %Collecting f values
0204         FhTerm = zeros(Nnze,2);
0205         FTerm = zeros(Nnze,2);
0206         if Options.Dependence.F<1
0207            FhTerm = ones(Nnze,2)*ValF;
0208            FTerm = FhTerm;
0209            FhTheta = FhTerm(:,1);
0210         else
0211            switch (Options.Dependence.F)
0212                case 1
0213                   FhTerm(:,2) = ones(Nnze,1)*ValF(2);
0214                   FTerm(:,2) = FhTerm(:,2);
0215                   if BackwardValue
0216                       FhTerm(:,1) = ones(Nnze,1);
0217                       FTerm(:,1) = FhTerm(:,1);
0218                       FhTheta = Theta*FhTerm(:,2) + (1-Theta)*FhTerm(:,2);
0219                   else
0220                       FhTheta = FhTerm(:,2);
0221                   end
0222                case 2
0223                   FhTerm(:,2) = QBasisfct(j,:)*...
0224                                              [Projection.F(Nodes(:,1),1),...
0225                                               Projection.F(Nodes(:,2),1),...
0226                                               Projection.F(Nodes(:,3),1)]';
0227                   FTerm(:,2) = feval(Driver,'f(x,t)',QuadraturVertices);
0228                   FhTheta = FhTerm(:,2);
0229                case 3
0230                   FhTerm(:,2) = QBasisfct(j,:)*...
0231                                              [Projection.F(Nodes(:,1),2),...
0232                                               Projection.F(Nodes(:,2),2),...
0233                                               Projection.F(Nodes(:,3),2)]';
0234                   FTerm(:,2) = feval(Driver,'f(x,t)',QuadraturVertices,T);
0235                   if BackwardValue
```

```
0236                    FhTerm(:,1) = QBasisfct(j,:)*...
0237                [Projection.F(Nodes(:,1),1) Projection.F(Nodes(:,2),1),...
0238                                            Projection.F(Nodes(:,3),1)]';
0239                    FTerm(:,1) = ...
0240                            feval(Driver,'f(x,t)',QuadraturVertices,Told);
0241                    FhTheta = Theta*FhTerm(:,2) + (1-Theta)*FhTerm(:,1);
0242                else
0243                    FhTheta = FhTerm(:,2);
0244                end
0245            end
0246        end
0247
0248        %Collecting divD values
0249        if Options.Dependence.D<2
0250            DivDTerm = zeros(Nnze,1);
0251        else
0252             DivDTerm = Theta*...
0253           dot(feval(Driver,'derD(x,t)',QuadraturVertices,T),NablaUTheta,2);
0254            if BackwardValue
0255                DivDTerm = DivDTerm + (1-Theta)*...
0256        dot(feval(Driver,'derD(x,t)',QuadraturVertices,Told),NablaUTheta,2);
0257            end
0258        end
0259
0260        %Collecting c values
0261        if Options.Dependence.C<1
0262            ChTerm = NablaUTheta*ValC';
0263            CTerm = ChTerm;
0264        elseif Options.Dependence.C == 1
0265             ChTerm =NablaUTheta*ValC(2,:)'+(1-Theta)*NablaUTheta*ValC(1,:)';
0266             CTerm = ChTerm;
0267        else
0268           a = [QBasisfct(j,:)*[Projection.C(Nodes(:,1),1), ...
0269                                Projection.C(Nodes(:,2),1), ...
0270                                Projection.C(Nodes(:,3),1)]';...
0271                QBasisfct(j,:)*[Projection.C(Nodes(:,1),2), ...
0272                                Projection.C(Nodes(:,2),2), ...
0273                                Projection.C(Nodes(:,3),2)]'];
0274           ChTerm  = dot(a',NablaUTheta,2);
0275            CTerm = Theta*...
0276              dot(feval(Driver,'c(x,t)',QuadraturVertices,T),NablaUTheta,2);
0277           if BackwardValue
0278                CTerm =  CTerm + (1-Theta)*...
0279           dot(feval(Driver,'c(x,t)',QuadraturVertices,Told),NablaUTheta,2);
0280            end
0281        end
0282
0283        %Collecting r values
0284        if Options.Dependence.R<1
```

```
0285            RhTerm = (ValR*Options.QBasisfct(j,:)*...
0286                                                [UTheta1 UTheta2 UTheta3]')';
0287            RTerm = RhTerm;
0288        elseif Options.Dependence.R == 1
0289             RhTerm = (Theta*ValR(2)+(1-Theta)*ValR(1))*(QBasisfct(j,:)*...
0290                                                [UTheta1 UTheta2 UTheta3]')';
0291             RTerm = RhTerm;
0292        else
0293            RhTerm = (QBasisfct(j,:)*[Projection.R(Nodes(:,1)).*UTheta1, ...
0294                                      Projection.R(Nodes(:,2)).*UTheta2,...
0295                                      Projection.R(Nodes(:,3)).*UTheta3]')';
0296            RTerm = Theta*feval(Driver,'r(x,t)',QuadraturVertices,T);
0297           if BackwardValue
0298               RTerm =  RTerm + (1-Theta)*...
0299                             feval(Driver,'r(x,t)',QuadraturVertices,Told);
0300           end
0301           RTerm = RTerm.*(QBasisfct(j,:)*[UTheta1 UTheta2 UTheta3]')';
0302        end
0303
0304        %Collect u_h^n-u_h^n-1
0305        DiffU = (QBasisfct(j,:)*[U1-U1old U2-U2old U3-U3old]')';
0306
0307        %Element residual
0308        a = FhTheta - DiffU/LocTau + DivDhTerm - ChTerm - RhTerm;
0309         ElementResiduals =  ElementResiduals + QWeights(j)*DoubleArea.*a.^2;
0310
0311        %F-data error estimator
0312        a = Theta^2*(FTerm(:,2)-FhTerm(:,2)).^2;
0313        if BackwardValue
0314            a = a + (1-Theta)^2*(FTerm(:,1)-FhTerm(:,1)).^2;
0315        end
0316        FDataErrorEstimators = FDataErrorEstimators + ...
0317                                                  QWeights(j)*DoubleArea.*a;
0318
0319        %data residuals
0320        a = -DivDhTerm + DivDTerm + ChTerm - CTerm + RhTerm - RTerm;
0321        DataResiduals = DataResiduals + QWeights(j)*DoubleArea.*a.^2;
0322
0323    end
0324
0325    %obtain edge flow
0326    EdgeLengths = sqrt([(B(:,1)-A(:,1)).^2 + (B(:,2)-A(:,2)).^2 ,...
0327                        (C(:,1)-B(:,1)).^2 + (C(:,2)-B(:,2)).^2 ,...
0328                        (A(:,1)-C(:,1)).^2 + (A(:,2)-C(:,2)).^2]);
0329    Normalvectors = ...
0330                   [(Rotation*(B-A)')'./(EdgeLengths(:,1)*ones(1,2)) ,...
0331                    (Rotation*(C-B)')'./(EdgeLengths(:,2)*ones(1,2)) ,...
0332                    (Rotation*(A-C)')'./(EdgeLengths(:,3)*ones(1,2))];
0333
```

```
0334   ValuesFlowh = zeros(Nnze,3);
0335   ValuesFlow = zeros(Nnze,3);
0336   for i=1:3
0337   %Collecting D values
0338       if Options.Dependence.D == 0
0339            DhEdge = ValD;
0340            DEdge =  ValD;
0341            ValuesFlowh(:,i) = ...
0342              -dot(Normalvectors(:,2*i-1:2*i),(DhEdge*NablaUTheta')',2);
0343            ValuesFlow(:,i) = ...
0344               -dot(Normalvectors(:,2*i-1:2*i),(DEdge*NablaUTheta')',2);
0345       elseif Options.Dependence.D == 1
0346             DhEdge = Theta*ValD(3:4,:) + (1-Theta)*ValD(1:2,:);
0347             DEdge =  DhEdge;
0348             ValuesFlowh(:,i) = ...
0349              -dot(Normalvectors(:,2*i-1:2*i),(DhEdge*NablaUTheta')',2);
0350             ValuesFlow(:,i) = ...
0351               -dot(Normalvectors(:,2*i-1:2*i),(DEdge*NablaUTheta')',2);
0352       else
0353             DhEdge = (Projection.D(Nodes(:,i),:,:)+...
0354                       Projection.D(Nodes(:,mod(i,3)+1),:,:))/2;
0355             DhEdge = reshape(permute(DhEdge,[2 1 3]),2*size(A,1),2);
0356             DhEdge = blktridiag(DhEdge,size(A,1));
0357             DEdge = Theta*feval(Driver,'D(x,t)',...
0358                         (CommonGrid.C4N(Nodes(:,i),:)+...
0359                          CommonGrid.C4N(Nodes(:,mod(i,3)+1),:))/2,T);
0360             if BackwardValue
0361                   DEdge = DEdge + (1-Theta)*feval(Driver,'D(x,t)',...
0362                         (CommonGrid.C4N(Nodes(:,i),:)+...
0363                          CommonGrid.C4N(Nodes(:,mod(i,3)+1),:))/2,T);
0364             end
0365             DEdge = reshape(permute(DEdge,[2 1 3]),2*size(A,1),2);
0366             DEdge = blktridiag(DEdge,size(A,1));
0367             ValuesFlowh(:,i) = -dot(Normalvectors(:,2*i-1:2*i),...
0368                  reshape(DhEdge*reshape(NablaUTheta',2*size(A,1),1),...
0369                                                  2,size(A,1))',2);
0370             ValuesFlow(:,i) = -dot(Normalvectors(:,2*i-1:2*i),...
0371                  reshape(DEdge*reshape(NablaUTheta',2*size(A,1),1),...
0372                                                  2,size(A,1))',2);
0373       end
0374
0375    end
0376    ValuesLength = reshape(EdgeLengths',3*Nnze,1);
0377    ValuesFlowh = reshape(ValuesFlowh',3*Nnze,1);
0378    ValuesFlow = reshape(ValuesFlow',3*Nnze,1);
0379
0380   %Assembly of flow matrices
0381    Flowh = ...
0382    sparse(I1,I2,ValuesFlowh,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
```

```
0383     Flow = ...
0384      sparse(I1,I2,ValuesFlow,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0385     Length = ...
0386    sparse(I1,I2,ValuesLength,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0387
0388    %Delete Dirichlet edges flow
0389    for i=1:size(CommonGrid.N4D,1)
0390        Flowh(CommonGrid.N4D(i,1),CommonGrid.N4D(i,2)) = 0;
0391        Flow(CommonGrid.N4D(i,1),CommonGrid.N4D(i,2)) = 0;
0392    end
0393
0394    %Data edge residuals
0395    Integrand = (Flowh - Flow + Flowh' - Flow' ).^2;
0396    DataEdgeResidual = Length .* Integrand;
0397
0398    %Neumann edges flow
0399    LocG = zeros(2,1);
0400    if ~isempty(CommonGrid.N4N)
0401         if Options.Dependence.G<1
0402             for i=1:size(CommonGrid.N4N,1)
0403                 Flowh(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) = ValG;
0404                 Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) = ...
0405                           norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)...
0406                                -CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0407             end
0408         else
0409             switch (Options.Dependence.G)
0410                 case 1
0411                     for i=1:size(CommonGrid.N4N,1)
0412                         LocG =Theta*G(2)+(1-Theta)*G(1);
0413                         Flowh(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) = ...
0414                                                                          LocG;
0415                         Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) =...
0416                               norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)-...
0417                                      CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0418                         GCon = ...
0419                           Theta^2*(G(2) - feval(Driver,'g(x,t)',[0 0],T))^2;
0420                         if BackwardValue
0421                             GCon = GCon + (1-Theta)^2 ...
0422                               *(G(1) - feval(Driver,'g(x,t)',[0 0],Told))^2;
0423                         end
0424                         GContribution(CommonGrid.N4N(i,1),...
0425                                                 CommonGrid.N4N(i,2)) =...
0426                        Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1))*GCon;
0427                     end
0428                 case 2
0429                     for i=1:size(CommonGrid.N4N,1)
0430                         LocG = (Projection.G(CommonGrid.N4N(i,1))...
0431                                 + Projection.G(CommonGrid.N4N(i,2)))/2;
```

```
0432                        Flowh(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) =...
0433                                                                    LocG;
0434                        Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) =...
0435                              norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)-...
0436                                    CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0437                        GContribution(CommonGrid.N4N(i,1),...
0438                                                 CommonGrid.N4N(i,2)) = ...
0439                          Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1))...
0440                            *(LocG - feval(Driver,'g(x,t)',...
0441                              (CommonGrid.C4N(CommonGrid.N4N(i,1),:)+...
0442                               CommonGrid.C4N(CommonGrid.N4N(i,2),:))/2))^2;
0443                    end
0444                case 3
0445                    for i=1:size(CommonGrid.N4N,1)
0446                        QuadraturVertices = ...
0447                                  (CommonGrid.C4N(CommonGrid.N4N(i,1),:)+...
0448                                   CommonGrid.C4N(CommonGrid.N4N(i,2),:))/2;
0449                        LocG(2) = ...
0450                               Theta*(Projection.G(CommonGrid.N4N(i,1),2)...
0451                                   + Projection.G(CommonGrid.N4N(i,2),2))/2;
0452                        GCon = Theta^2*(LocG(2)-feval(Driver,'g(x,t)',...
0453                                                 QuadraturVertices,T))^2;
0454                        if BackwardValue
0455                            LocG(1) = (1-Theta)...
0456                                    *(Projection.G(CommonGrid.N4N(i,1),1)...
0457                                    + Projection.G(CommonGrid.N4N(i,2),1))/2;
0458                            GCon = GCon ...
0459                            + (1-Theta)^2*(LocG(1)-feval(Driver,'g(x,t)',...
0460                                                 QuadraturVertices,Told))^2;
0461                        end
0462                        Flowh(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) = ...
0463                                              Theta*LocG(2)+(1-Theta)*LocG(1);
0464                        Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1)) =...
0465                              norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)-...
0466                                    CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0467                        GContribution(CommonGrid.N4N(i,1),...
0468                                                 CommonGrid.N4N(i,2)) = ...
0469                          Length(CommonGrid.N4N(i,2),CommonGrid.N4N(i,1))...
0470                                                                    *GCon;
0471                    end
0472            end
0473        end
0474    end
0475
0476    %Edge residuals
0477    Integrand = (Flowh + Flowh').^2;
0478    EdgeResidual = Length .* Integrand;
0479
0480    %EdgeCounter considers edges at the boundary by value 1, other edges by
```

```
0481    %value 1/2
0482    EdgeCounter = ...
0483              sparse(I1,I2,1,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0484    EdgeCounter = EdgeCounter - EdgeCounter'/2;
0485
0486
0487    %Compute local indicators, and element part of spatial and
0488    %data indicators
0489    Diameters = max(EdgeLengths,[],2);
0490    if Beta
0491        AlphaK = min([Diameters*Lambda^(-1/2),Beta^(-1/2)*ones(Nnze,1)],[],2);
0492    else
0493        AlphaK = Diameters*Lambda^(-1/2);
0494    end
0495    VerfurthErrorEstimator = sum(AlphaK.^2.*ElementResiduals);
0496    FDataErrorEstimator = sum(AlphaK.^2.*FDataErrorEstimators);
0497    DataErrorEstimator = sum(AlphaK.^2.*DataResiduals);
0498    RefinementIndicator = AlphaK.^2.*...
0499                      (ElementResiduals+DataResiduals+FDataErrorEstimators);
0500
0501    %edges
0502    %Obtain AlphaE
0503    ValAlphaE = zeros(Nnze,3);
0504    for i=1:3
0505       if Beta
0506          ValAlphaE(:,i) = min([EdgeLengths(:,i)*Lambda^(-1/2),...
0507                                              Beta^(-1/2)*ones(Nnze,1)],[],2);
0508       else
0509          ValAlphaE(:,i) = EdgeLengths(:,i)*Lambda^(-1/2);
0510       end
0511    end
0512    ValAlphaE = reshape(ValAlphaE',3*Nnze,1);
0513    AlphaE = ...
0514      sparse(I1,I2,ValAlphaE,size(CommonGrid.C4N,1),size(CommonGrid.C4N,1));
0515
0516    LocEdgeContribution = Lambda^(-1/2)*AlphaE.*...
0517                 (EdgeResidual + DataEdgeResidual + GContribution);
0518    GContributionLocal = Lambda^(-1/2)*sum(sum(AlphaE.*GContribution));
0519
0520    %refinement indicators
0521    RefinementIndicator = RefinementIndicator + ...
0522                extractValues(LocEdgeContribution,Nodes(:,1),Nodes(:,2))+...
0523                extractValues(LocEdgeContribution,Nodes(:,2),Nodes(:,3))+...
0524                extractValues(LocEdgeContribution,Nodes(:,3),Nodes(:,1));
0525
0526    %Verfuerth-type error estimator
0527    VerfurthErrorEstimator = VerfurthErrorEstimator ...
0528                + Lambda^(-1/2)*sum(sum(AlphaE.*EdgeResidual.*EdgeCounter));
0529    DataErrorEstimator = DataErrorEstimator ...
```

```
0530            + Lambda^(-1/2)*sum(sum(AlphaE.*DataEdgeResidual.*EdgeCounter));
0531
0532
0533    %Arrange contributions
0534    EtaSpatial.VerfuerthErrorEstimator = sqrt(VerfurthErrorEstimator);
0535    EtaSpatial.DataErrorEstimator = sqrt(DataErrorEstimator);
0536    EtaSpatial.FDataErrorEstimator = sqrt(FDataErrorEstimator);
0537    EtaSpatial.NeumannErrorEstimator = sqrt(GContributionLocal);
0538
0539    %Sum up contributions
0540    EtaSpatial.SpatialErrorEstimator = ...
0541                                  (EtaSpatial.VerfuerthErrorEstimator^2 +...
0542                                   EtaSpatial.DataErrorEstimator^2 +...
0543                                   EtaSpatial.FDataErrorEstimator^2 + ...
0544                                   EtaSpatial.NeumannErrorEstimator^2)^(1/2);
0545
0546
0547    %mark elements
0548    if ~nnz(EtaSpatial.SpatialErrorEstimator)
0549       %special case of too little freenodes
0550       I1 = 1:size(CommonGrid.N4E,1);
0551       I3 = [];
0552    else
0553       %normal case
0554       I1 = find(RefinementIndicator>=...
0555                            max(RefinementIndicator)*Options.IndicatorTheta);
0556       I3 = find(RefinementIndicator>=...
0557                          median(RefinementIndicator)*Options.IndicatorTheta);
0558
0559     end
0560
0561    %From CommonGrid to Grid
0562    I1 = CommonGrid.N4E(nze(I1),4);
0563    I1 = unique(I1);
0564    I3 = setdiff(1:size(Grid.N4E,1),CommonGrid.N4E(nze(I3),4));
0565    I2 = ones(nnz(I1),1);
0566    I3 = setdiff(I3,I1);
0567    I2 = [I2; 2*ones(nnz(I3),1)];
0568    I1 = [I1;I3'];
0569    MarkedElements = sparse(I1,I2,1,size(Grid.N4E,1),2);
0570
0571    %Finalizing
0572    LogStruct.ComputationTime = cputime - CPU_time;
0573
0574    if Options.ShowTime
0575        ComputationTimeIndicator = cputime - CPU_time
0576    end
0577
0578    if Options.DisplayOutput
```

```
0579        tricontour(CommonGrid.N4E(nze,1:3),...
0580                                        CommonGrid.C4N,RefinementIndicator(nze));
0581    end
0582
0583   %end of etaSpatial
0584 %-------------------------------------------------------------------------
0585 function [Projection,DoubleArea] =l2Projections(Grid,Told,T,Driver,Options)
0586 %l2Projections: L2-projections for data functions f,g,D,c,r
0587
0588    Theta =Options.Theta;
0589    %Renaming for readability
0590    N4E = Grid.N4E;
0591    C4N = Grid.C4N;
0592    N4N = Grid.N4N;
0593    clear Grid.N4E Grid.C4N Grid.N4N
0594
0595    %Initialization
0596    nze = find(N4E(:,1));
0597    nzc = setdiff((1:size(C4N,1))',Grid.Unused.C4N);
0598
0599    if Theta==1
0600        BackwardValue = false;
0601    else
0602        BackwardValue = true;
0603    end
0604
0605    %Prelocating vectors for assembly
0606    a = [N4E(nze,1) N4E(nze,1) N4E(nze,1) N4E(nze,2) N4E(nze,2)...
0607    N4E(nze,2) N4E(nze,3) N4E(nze,3) N4E(nze,3)]';
0608    I2 = reshape(a,9*nnz(nze),1);
0609    a = [N4E(nze,1:3) N4E(nze,1:3) N4E(nze,1:3)]';
0610    I1 = reshape(a,9*nnz(nze),1);
0611    I3 = reshape(N4E(nze,1:3)',3*nnz(nze),1);
0612    %Assembly of mass matrix for Grid
0613    A = C4N(N4E(nze,1),:);
0614    B = C4N(N4E(nze,2),:);
0615    C = C4N(N4E(nze,3),:);
0616
0617    if isempty(Grid.M)
0618
0619       %Assembly of mass matrix for Grid
0620       A = C4N(N4E(nze,1),:);
0621       B = C4N(N4E(nze,2),:);
0622       C = C4N(N4E(nze,3),:);
0623       %Assembly of areas
0624       DoubleArea = B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
0625                    -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0626       %Assembly of mass matrix
0627       RefM = [2 1 1 1 2 1 1 1 2]/24;
```

```
0628      LocMassMatrices = reshape(RefM'*DoubleArea',9*size(A,1),1);
0629      M = sparse(I1,I2,LocMassMatrices,size(C4N,1),size(C4N,1));
0630   else
0631      %load stored data
0632      M = Grid.M;
0633      DoubleArea = Grid.DoubleArea;
0634   end
0635   N = M;
0636   M = M(nzc,nzc);
0637
0638   if ~isempty(find(diag(M)==0))
0639       %repair of mass matrix necessary
0640       save debug2 M Grid
0641       a = [N4E(nze,1); N4E(nze,2); N4E(nze,3)];
0642       nzc = intersect(a,nzc);
0643       M = N(nzc,nzc);
0644       if ~isempty(find(diag(M)==0))
0645            error('Failed to repair mass matrix')
0646        end
0647   end
0648
0649   if Options.LargeDimensionMode
0650      save temp2 M;
0651   end
0652
0653   %L2-projection of f^{(n),\theta}
0654   Projection.F = zeros(size(C4N,1),1);
0655   switch (Options.Dependence.F)
0656
0657       case 2
0658
0659          Values = zeros(3,size(A,1));
0660          for i=1:size(Options.QWeights,1)
0661              Values = Values + ...
0662                              Options.QWeights(i)*Options.QBasisfct(i,:)'*...
0663           (feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(i,1)+...
0664                                  (C-A)*Options.QNodes(i,2),0).*DoubleArea)';
0665          end
0666          Values = reshape(Values,3*size(A,1),1);
0667          RHS = sparse(I3,ones(size(I3,1),1),Values,size(C4N,1),1);
0668          RHS2 = RHS(nzc);
0669          if Options.LargeDimensionMode
0670
0671             clear X
0672             save temp;
0673             save temp3 RHS2;
0674             clear all;
0675             load temp2;
0676             load temp3;
```

```
0677            X = M\RHS2;
0678            load temp
0679            Projection.F(nzc) = X;
0680         else
0681            Projection.F(nzc) = M\RHS2;
0682         end
0683
0684      case 3
0685         Projection.F = zeros(size(C4N,1),2);
0686         Values = zeros(3,size(A,1));
0687         Values2 = zeros(3,size(A,1));
0688         for i=1:size(Options.QWeights,1)
0689            Values = Values + ...
0690                           Options.QWeights(i)*Options.QBasisfct(i,:)'*...
0691             (feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(i,1)+...
0692                               (C-A)*Options.QNodes(i,2),T).*DoubleArea)';
0693            if BackwardValue
0694                Values2 = Values2 + ...
0695                           Options.QWeights(i)*Options.QBasisfct(i,:)'*...
0696       (feval(Driver,'f(x,t)',A+(B-A)*Options.QNodes(i,1)+...
0697                            (C-A)*Options.QNodes(i,2),Told).*DoubleArea)';
0698            end
0699         end
0700         Values = reshape(Values,3*size(A,1),1);
0701         RHS(:,2) = sparse(I3,ones(size(I3,1),1),Values,size(C4N,1),1);
0702         if BackwardValue
0703             Values2 = reshape(Values2,3*size(A,1),1);
0704             RHS(:,1)= sparse(I3,ones(size(I3,1),1),Values2,size(C4N,1),1);
0705             RHS(:,1)= (1-Options.Theta)*RHS(:,1);
0706         end
0707         RHS2 = RHS(nzc,:);
0708         RHS2(:,2) = Options.Theta*RHS2(:,2);
0709
0710         if Options.LargeDimensionMode
0711
0712            clear X
0713            save temp;
0714            save temp3 RHS2;
0715            clear all;
0716            load temp2;
0717            load temp3;
0718            X = M\RHS2;
0719            load temp
0720            Projection.F(nzc,:) = X;
0721         else
0722            Projection.F(nzc,:) = M\RHS2;
0723         end
0724   end
0725
```

```
0726   %L2-projection of g^{(n),\theta}
0727   if size(N4N,1) && (Options.Dependence.G>1)
0728
0729       N = sparse(size(C4N,1),size(C4N,1));
0730       for j=1:size(N4N,1)
0731            N(N4N(j,:),N4N(j,:)) = N(N4N(j,:),N4N(j,:)) ...
0732                        + norm(C4N(N4N(j,1),:)-C4N(N4N(j,2),:))*[2 1;1 2]/6;
0733       end
0734
0735       switch (Options.Dependence.G)
0736           case 2
0737              Projection.G = zeros(size(C4N,1),1);
0738              RHS = zeros(size(C4N,1),1);
0739              for j=1:size(N4N,1)
0740                  Vertices = C4N(N4N(j,1:2),:);
0741                  RHS(N4N(j,1:2)) = RHS(N4N(j,1:2)) ...
0742                    + norm(Vertices(1,:)-Vertices(2,:))*feval(Driver,...
0743                          'g(x,t)',(Vertices(1,:)-Vertices(2,:))/2)*[1;1]/2;
0744              end
0745
0746              a = unique(N4N);
0747              N = N(a,a);
0748              RHS2 = RHS(a);
0749              if Options.LargeDimensionMode
0750
0751                  clear X
0752                  save temp;
0753                  save temp2 N RHS2;
0754                  clear all;
0755                  load temp2;
0756                  X = N\RHS2;
0757                  load temp
0758                  Projection.G(a) = X;
0759              else
0760                  Projection.G(a) = N\RHS2;
0761              end
0762           case 3
0763              Projection.G = zeros(size(C4N,1),2);
0764              RHS = zeros(size(C4N,1),2);
0765              for j=1:size(N4N,1)
0766                  Vertices = C4N(N4N(j,1:2),:);
0767                  a = feval(Driver,'g(x,t)',sum(Vertices)/2,T);
0768                  RHS(N4N(j,1:2),2) = RHS(N4N(j,1:2),2)...
0769                            + norm(Vertices(1,:)-Vertices(2,:))*a/2;
0770                  if BackwardValue
0771                     b = feval(Driver,'g(x,t)',sum(Vertices)/2,Told);
0772                     RHS(N4N(j,1:2),1) = RHS(N4N(j,1:2),1)...
0773                            + norm(Vertices(1,:)-Vertices(2,:))*b/2;
0774                  end
```

```
0775              end
0776             a = unique(N4N);
0777             N = N(a,a);
0778             RHS2 = RHS(a,:);
0779             if Options.LargeDimensionMode
0780
0781                 clear X
0782                 save temp;
0783                 save temp2 N RHS2;
0784                 clear all;
0785                 load temp2;
0786                 X = N\RHS2;
0787                 load temp
0788                 Projection.G(a,:) = X;
0789             else
0790                 Projection.G(a,:) = N\RHS2;
0791             end
0792     end
0793 end
0794
0795 %L2-projection of D^{(n),\theta}
0796 switch (Options.Dependence.D)
0797     case 2
0798        Projection.D = zeros(size(C4N,1),2,2);
0799        RHS = sparse(size(C4N,1),4);
0800        for j=1:size(Options.QWeights,1)
0801           Values = Options.QWeights(j)*feval(Driver,'D(x,t)',...
0802                 A+(B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2),0);
0803           Values = [Options.QBasisfct(j,1)*Values,...
0804                     Options.QBasisfct(j,2)*Values,...
0805                     Options.QBasisfct(j,3)*Values];
0806           Values = Values.*repmat(DoubleArea,2,6);
0807           Values = [reshape([Values(1:size(A,1),1),...
0808                              Values(1:size(A,1),3),...
0809                              Values(1:size(A,1),5)]',3*size(A,1),1),...
0810                     reshape([Values(size(A,1)+1:end,1),...
0811                              Values(size(A,1)+1:end,3),...
0812                              Values(size(A,1)+1:end,5)]',3*size(A,1),1),...
0813                     reshape([Values(1:size(A,1),2),...
0814                              Values(1:size(A,1),4),...
0815                              Values(1:size(A,1),6)]',3*size(A,1),1),...
0816                     reshape([Values(size(A,1)+1:end,2),...
0817                              Values(size(A,1)+1:end,4),...
0818                              Values(size(A,1)+1:end,6)]',3*size(A,1),1)];
0819           RHS(:,1) = RHS(:,1) + ...
0820                     sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
0821           RHS(:,2) = RHS(:,2) + ...
0822                     sparse(I3,ones(nnz(I3),1),Values(:,2),size(C4N,1),1);
0823           RHS(:,3) = RHS(:,3) + ...
```

```
0824                    sparse(I3,ones(nnz(I3),1),Values(:,3),size(C4N,1),1);
0825            RHS(:,4) = RHS(:,4) + ...
0826                    sparse(I3,ones(nnz(I3),1),Values(:,4),size(C4N,1),1);
0827        end
0828        RHS2 = RHS(nzc,1);
0829        if Options.LargeDimensionMode
0830
0831            clear X
0832            save temp;
0833            save temp3 RHS2;
0834            clear all;
0835            load temp2;
0836            load temp3;
0837            X = M\RHS2;
0838            load temp
0839            Projection.D(nzc,1,1) = X;
0840        else
0841            Projection.D(nzc,1,1) = M\RHS2;
0842        end
0843
0844        RHS2 = RHS(nzc,2);
0845        if Options.LargeDimensionMode
0846
0847            clear X
0848            save temp;
0849            save temp3 RHS2;
0850            clear all;
0851            load temp2;
0852            load temp3;
0853            X = M\RHS2;
0854            load temp
0855            Projection.D(nzc,2,1) = X;
0856        else
0857            Projection.D(nzc,2,1) = M\RHS2;
0858        end
0859
0860        RHS2 = RHS(nzc,3);
0861        if Options.LargeDimensionMode
0862
0863            clear X
0864            save temp;
0865            save temp3 RHS2;
0866            clear all;
0867            load temp2;
0868            load temp3;
0869            X = M\RHS2;
0870            load temp
0871            Projection.D(nzc,1,2) = X;
0872        else
```

```
0873            Projection.D(nzc,1,2) = M\RHS2;
0874         end
0875
0876         RHS2 = RHS(nzc,4);
0877         if Options.LargeDimensionMode
0878
0879            clear X
0880            save temp;
0881            save temp3 RHS2;
0882            clear all;
0883            load temp2;
0884            load temp3;
0885            X = M\RHS2;
0886            load temp
0887            Projection.D(nzc,2,2) = X;
0888         else
0889            Projection.D(nzc,2,2) = M\RHS2;
0890         end
0891
0892      case 3
0893         Projection.D = zeros(size(C4N,1),2,2);
0894         RHS = sparse(size(C4N,1),4);
0895         for j=1:size(Options.QWeights,1)
0896            Values = Options.Theta*Options.QWeights(j)*...
0897                     feval(Driver,'D(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0898                                           (C-A)*Options.QNodes(j,2),T);
0899            if BackwardValue
0900               Values = Values + (1-Options.Theta)*Options.QWeights(j)...
0901                   *feval(Driver,'D(x,t)',A+(B-A)*Options.QNodes(j,1)+...
0902                                          (C-A)*Options.QNodes(j,2),Told);
0903            end
0904            Values = [Options.QBasisfct(j,1)*Values,...
0905                      Options.QBasisfct(j,2)*Values,...
0906                      Options.QBasisfct(j,3)*Values];
0907            Values = Values.*repmat(DoubleArea,2,6);
0908            Values = [reshape([Values(1:size(A,1),1),...
0909                               Values(1:size(A,1),3),...
0910                               Values(1:size(A,1),5)]',3*size(A,1),1),...
0911                      reshape([Values(size(A,1)+1:end,1),...
0912                               Values(size(A,1)+1:end,3),...
0913                               Values(size(A,1)+1:end,5)]',3*size(A,1),1),...
0914                        reshape([Values(1:size(A,1),2),...
0915                                 Values(1:size(A,1),4),...
0916                                 Values(1:size(A,1),6)]',3*size(A,1),1),...
0917                        reshape([Values(size(A,1)+1:end,2),...
0918                                 Values(size(A,1)+1:end,4),...
0919                                 Values(size(A,1)+1:end,6)]',3*size(A,1),1)];
0920            RHS(:,1) = RHS(:,1) + ...
0921                       sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
```

```
0922            RHS(:,2) = RHS(:,2) + ...
0923                      sparse(I3,ones(nnz(I3),1),Values(:,2),size(C4N,1),1);
0924            RHS(:,3) = RHS(:,3) + ...
0925                      sparse(I3,ones(nnz(I3),1),Values(:,3),size(C4N,1),1);
0926            RHS(:,4) = RHS(:,4) + ...
0927                      sparse(I3,ones(nnz(I3),1),Values(:,4),size(C4N,1),1);
0928        end
0929
0930        RHS2 = RHS(nzc,1);
0931        if Options.LargeDimensionMode
0932
0933            clear X
0934            save temp;
0935            save temp3 RHS2;
0936            clear all;
0937            load temp2;
0938            load temp3;
0939            X = M\RHS2;
0940            load temp
0941            Projection.D(nzc,1,1) = X;
0942        else
0943            Projection.D(nzc,1,1) = M\RHS2;
0944        end
0945
0946        RHS2 = RHS(nzc,2);
0947        if Options.LargeDimensionMode
0948
0949            clear X
0950            save temp;
0951            save temp3 RHS2;
0952            clear all;
0953            load temp2;
0954            load temp3;
0955            X = M\RHS2;
0956            load temp
0957            Projection.D(nzc,2,1) = X;
0958        else
0959            Projection.D(nzc,2,1) = M\RHS2;
0960        end
0961
0962        RHS2 = RHS(nzc,3);
0963        if Options.LargeDimensionMode
0964
0965            clear X
0966            save temp;
0967            save temp3 RHS2;
0968            clear all;
0969            load temp2;
0970            load temp3;
```

```
0971              X = M\RHS2;
0972              load temp
0973              Projection.D(nzc,1,2) = X;
0974          else
0975              Projection.D(nzc,1,2) = M\RHS2;
0976          end
0977
0978          RHS2 = RHS(nzc,4);
0979          if Options.LargeDimensionMode
0980
0981              clear X
0982              save temp;
0983              save temp3 RHS2;
0984              clear all;
0985              load temp2;
0986              load temp3;
0987              X = M\RHS2;
0988              load temp
0989              Projection.D(nzc,2,2) = X;
0990          else
0991              Projection.D(nzc,2,2) = M\RHS2;
0992          end
0993    end
0994
0995    %L2-projection of c^{(n),\theta}
0996    switch (Options.Dependence.C)
0997        case 2
0998           Projection.C = zeros(size(C4N,1),2);
0999           RHS = sparse(size(C4N,1),2);
1000           for j=1:size(Options.QWeights,1)
1001              Values = Options.QWeights(j)*...
1002                       feval(Driver,'c(x,t)',A+(B-A)*Options.QNodes(j,1)+...
1003                                             (C-A)*Options.QNodes(j,2),0);
1004              Values = [Options.QBasisfct(j,1)*Values,...
1005                        Options.QBasisfct(j,2)*Values,...
1006                        Options.QBasisfct(j,3)*Values];
1007              Values = Values.*repmat(DoubleArea,1,6);
1008              Values = [reshape([Values(:,1),Values(:,3),Values(:,5)]',...
1009                                                        3*size(A,1),1),...
1010                        reshape([Values(:,2),Values(:,4),Values(:,6)]',...
1011                                                          3*size(A,1),1)];
1012              RHS(:,1) = RHS(:,1) + ...
1013                        sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
1014              RHS(:,2) = RHS(:,2) + ...
1015                        sparse(I3,ones(nnz(I3),1),Values(:,2),size(C4N,1),1);
1016           end
1017
1018           RHS2 = RHS(nzc,:);
1019           if Options.LargeDimensionMode
```

```
1020
1021              clear X
1022              save temp;
1023              save temp3 RHS2;
1024              clear all;
1025              load temp2;
1026              load temp3;
1027              X = M\RHS2;
1028              load temp
1029              Projection.C(nzc,:) = X;
1030           else
1031              Projection.C(nzc,:) = M\RHS2;
1032           end
1033        case 3
1034           Projection.C = zeros(size(C4N,1),2);
1035           RHS = sparse(size(C4N,1),2);
1036           for j=1:size(Options.QWeights,1)
1037              Values = Options.Theta*Options.QWeights(j)*...
1038                       feval(Driver,'c(x,t)',A+(B-A)*Options.QNodes(j,1)+...
1039                                             (C-A)*Options.QNodes(j,2),T);
1040              if BackwardValue
1041                  Values = Values + (1-Options.Theta)*Options.QWeights(j)...
1042                      *feval(Driver,'c(x,t)',A+(B-A)*Options.QNodes(j,1)+...
1043                                             (C-A)*Options.QNodes(j,2),Told);
1044              end
1045              Values = [Options.QBasisfct(j,1)*Values,...
1046                        Options.QBasisfct(j,2)*Values,...
1047                        Options.QBasisfct(j,3)*Values];
1048              Values = Values.*repmat(DoubleArea,1,6);
1049              Values = [reshape([Values(:,1),Values(:,3),Values(:,5)]',...
1050                                                        3*size(A,1),1),...
1051                        reshape([Values(:,2),Values(:,4),Values(:,6)]',...
1052                                                          3*size(A,1),1)];
1053              RHS(:,1) = RHS(:,1) + ...
1054                        sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
1055              RHS(:,2) = RHS(:,2) + ...
1056                        sparse(I3,ones(nnz(I3),1),Values(:,2),size(C4N,1),1);
1057           end
1058
1059           RHS2 = RHS(nzc,:);
1060           if Options.LargeDimensionMode
1061
1062              clear X
1063              save temp;
1064              save temp3 RHS2;
1065              clear all;
1066              load temp2;
1067              load temp3;
1068              X = M\RHS2;
```

```
1069             load temp
1070             Projection.C(nzc,:) = X;
1071         else
1072             Projection.C(nzc,:) = M\RHS2;
1073         end
1074   end
1075
1076   %L2-projection of r^{(n),\theta}
1077   switch (Options.Dependence.R)
1078       case 2
1079          Projection.R = zeros(size(C4N,1),1);
1080          RHS = sparse(size(C4N,1),1);
1081          for j=1:size(Options.QWeights,1)
1082             Values = Options.QWeights(j)*...
1083                      feval(Driver,'r(x,t)',A+(B-A)*Options.QNodes(j,1)+...
1084                                            (C-A)*Options.QNodes(j,2),0);
1085             Values = [Options.QBasisfct(j,1)*Values,...
1086                      Options.QBasisfct(j,2)*Values,...
1087                      Options.QBasisfct(j,3)*Values];
1088             Values = Values.*repmat(DoubleArea,1,3);
1089             Values = reshape(Values',3*size(A,1),1);
1090             RHS = RHS + ...
1091                      sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
1092          end
1093          RHS2 = RHS(nzc);
1094          if Options.LargeDimensionMode
1095
1096              clear X
1097              save temp;
1098              save temp3 RHS2;
1099              clear all;
1100              load temp2;
1101              load temp3;
1102              X = M\RHS2;
1103              load temp
1104              Projection.R(nzc) = X;
1105          else
1106              Projection.R(nzc) = M\RHS2;
1107          end
1108       case 3
1109          Projection.R = zeros(size(C4N,1),1);
1110          RHS = sparse(size(C4N,1),1);
1111          for j=1:size(Options.QWeights,1)
1112             Values = Options.Theta*Options.QWeights(j)*...
1113                      feval(Driver,'r(x,t)',A+(B-A)*Options.QNodes(j,1)+...
1114                                            (C-A)*Options.QNodes(j,2),T);
1115             if BackwardValue
1116                 Values = Values + (1-Options.Theta)*Options.QWeights(j)...
1117                     *feval(Driver,'r(x,t)',A+(B-A)*Options.QNodes(j,1)+...
```

```
1118                                          (C-A)*Options.QNodes(j,2),Told);
1119            end
1120            Values = [Options.QBasisfct(j,1)*Values,...
1121                      Options.QBasisfct(j,2)*Values,...
1122                      Options.QBasisfct(j,3)*Values];
1123            Values = Values.*repmat(DoubleArea,1,3);
1124            Values = reshape(Values',3*size(A,1),1);
1125            RHS = RHS + ...
1126                      sparse(I3,ones(nnz(I3),1),Values(:,1),size(C4N,1),1);
1127         end
1128
1129         RHS2 = RHS(nzc);
1130         if Options.LargeDimensionMode
1131
1132             clear X
1133             save temp;
1134             save temp3 RHS2;
1135             clear all;
1136             load temp2;
1137             load temp3;
1138             X = M\RHS2;
1139             load temp
1140             Projection.R(nzc) = X;
1141         else
1142             Projection.R(nzc) = M\RHS2;
1143         end
1144   end
1145
1146   %end of l2Projections
1147 %-------------------------------------------------------------------------
1148 function UCommon = getCommonU(OldGrid,Grid,CommonGrid,Uold,U)
1149 %getCommonU:obtaining values U,Uold at the vertices of CommonGrid by linear
1150 %interpolation
1151
1152     nzec = find(CommonGrid.N4E(:,1));
1153     A = CommonGrid.C4N(CommonGrid.N4E(nzec,1),:);
1154     B = CommonGrid.C4N(CommonGrid.N4E(nzec,2),:);
1155     C = CommonGrid.C4N(CommonGrid.N4E(nzec,3),:);
1156     Nodes = Grid.N4E(CommonGrid.N4E(nzec,4),1:3);
1157     OldNodes = OldGrid.N4E(CommonGrid.N4E(nzec,5),1:3);
1158     NewA = Grid.C4N(Nodes(:,1),:);
1159     NewB = Grid.C4N(Nodes(:,2),:);
1160     NewC = Grid.C4N(Nodes(:,3),:);
1161     OldA = OldGrid.C4N(OldNodes(:,1),:);
1162     OldB = OldGrid.C4N(OldNodes(:,2),:);
1163     OldC = OldGrid.C4N(OldNodes(:,3),:);
1164     U1 = U(Nodes(:,1));
1165     U2 = U(Nodes(:,2));
1166     U3 = U(Nodes(:,3));
```

```
1167     U1old = Uold(OldNodes(:,1));
1168     U2old = Uold(OldNodes(:,2));
1169     U3old = Uold(OldNodes(:,3));
1170     UCommon = zeros(size(CommonGrid.C4N,1),2);
1171
1172     %Obtain values of U on new grid
1173     M1 = reshape([NewB-NewA NewC-NewA]',2,2*size(NewA,1))';
1174     M1 = blktridiag(M1,size(NewA,1));
1175     CNodesRef = M1\reshape((A - NewA)',2*size(NewA,1),1);
1176     CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
1177     UCommon(CommonGrid.N4E(nzec,1),2)= ...
1178                 (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
1179                                     + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
1180     CNodesRef = M1\reshape((B - NewA)',2*size(NewA,1),1);
1181     CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
1182     UCommon(CommonGrid.N4E(nzec,2),2)= ...
1183                 (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
1184                                     + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
1185     CNodesRef = M1\reshape((C - NewA)',2*size(NewA,1),1);
1186     CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
1187     UCommon(CommonGrid.N4E(nzec,3),2)= ...
1188                 (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
1189                                     + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
1190     %Obtain values of U on old grid
1191     M2 = reshape([OldB-OldA OldC-OldA]',2,2*size(OldA,1))';
1192     M2 = blktridiag(M2,size(OldA,1));
1193     CNodesRef = M2\reshape((A - OldA)',2*size(OldA,1),1);
1194     CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
1195     UCommon(CommonGrid.N4E(nzec,1),1)= ...
1196              (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
1197                              + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
1198     CNodesRef = M2\reshape((B - OldA)',2*size(OldA,1),1);
1199     CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
1200     UCommon(CommonGrid.N4E(nzec,2),1)= ...
1201              (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
1202                              + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
1203     CNodesRef = M2\reshape((C - OldA)',2*size(OldA,1),1);
1204     CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
1205     UCommon(CommonGrid.N4E(nzec,3),1)= ...
1206              (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
1207                              + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
1208
1209     %end of getCommonU
1210
1211 %-------------------------------------------------------------------------
1212 function A = blktridiag(A,n)
1213 % blktridiag: computes a sparse (block) tridiagonal matrix with n blocks
1214
1215 [q,p] = size(A);
```

```
1216 [ind1,ind2,ind3]=ndgrid(0:p-1,0:p-1,0:n-1);
1217 rind = 1+ind1(:)+p*ind3(:);
1218 cind = 1+ind2(:)+p*ind3(:);
1219 v = reshape(A',n*4,1);
1220 A = sparse(rind,cind,v,n*p,n*p);
1221
1222   %end of blktridiag
1223 %---------------------------------------------------------------------------
1224 function Values = extractValues(A,I1,I2)
1225 %extractValues: extracts values of A indicated by I1 and I2 index vectors,
1226 %if A is too large for linear indexing, submatrices are constructed
1227
1228 Partitioner = floor(1e9/size(A,2));
1229 I3 = sub2ind(size(A),I1,I2);
1230 Values = zeros(size(I1,1),1);
1231 if max(I3)<1e9
1232     Values = A(I3);
1233 else
1234     for i=1:floor(size(A,1)/Partitioner)
1235
1236         a = find(I3>(i-1)*Partitioner*size(A,1));
1237         b = find(I3(a)<=i*Partitioner*size(A,1));
1238         I4 = I3(a(b));
1239         if ~isempty(I4)
1240             B = A(:,(i-1)*Partitioner+1:i*Partitioner);
1241             Values(a(b)) = ...
1242             Values(a(b)) ...
1243             + B(I4-(i-1)*Partitioner*size(A,1)*ones(length(I4),1));
1244         end
1245     end
1246
1247         a = find(I3>i*Partitioner*size(A,1));
1248         I4 = I3(a);
1249         if ~isempty(I4)
1250             B = A(:,i*Partitioner+1:end);
1251             Values(a) = ...
1252             Values(a) ...
1253             + B(I4-i*Partitioner*size(A,1)*ones(length(I4),1));
1254         end
1255 end
1256
1257   %end of extractValues
```

A.5 The Function *etaTemporal*

```
0001 function [EtaTemporal,LogStruct] = etaTemporal(Told,T,Driver,varargin)  %ok
0002 %etaTemproal: computes temporal error estimator
0003 %
0004 %   INPUT ARGUMENTS:
```

```
0005 %
0006 %   [MeanT: mean time between two time steps, scalar]
0007 %
0008 %   [Driver: name of the file, where the problem data is located; string]
0009 %
0010 %   [varargin:
0011 %         if nargin=2: varargin{1}=CommonGrid,varargin{2}=UCommon
0012 %         if nargin=6: varargin{1}=OldGrid,varargin{2}=Grid,
0013 %                                                    varargin{3}=Uold]
0014 %
0015 %   OUTPUT ARGUMENTS:
0016 %
0017 %   [EtaTemporal.Part1: |||Uold-U||| on CommonGrid]
0018 %
0019 %   [EtaTemporal.Part2: sum of ||mean values of time-derivatives of data||]
0020 %
0021 %   [EtaTemporal.TemporalErrorEstimator:
0022 %                        ~(EtaTemporal.Part1^2+EtaTemporal.Part2^2)^(1/2)]
0023 %
0024 %   [LogStruct is a log-struct which contains information about the
0025 %       solution process.]
0026 %
0027 %   See also commonGrid,parabolicSolver,etaSpatial
0028 %
0029 %   Copyright (c) 2007 Philipp Wissgott
0030 %                      Vienna University of Technology
0031
0032 %************************** OPTIONS ************************************
0033
0034   OptionFile = feval(Driver,'OptionFile');
0035   Options = feval(OptionFile,'LoadOptions','Indicator',Driver,0,1);
0036   Lambda = Options.Lambda;
0037   Beta = Options.Beta;
0038
0039 %************************** LOGGING ************************************
0040
0041   if nargout==3
0042
0043       Options.Logging = 1;
0044   else
0045       Options.Logging = 0;
0046   end
0047   CPU_time = cputime;
0048
0049 %************************ VARIABLE INPUT *******************************
0050
0051   if nargin==7
0052       try
0053          load CGrid
```

```
0054          CommonGrid = CGrid;
0055       catch
0056          CommonGrid = commonGrid(varargin{1},varargin{2},Driver);
0057       end
0058       UCommon = getCommonU(varargin{1},varargin{2},...
0059                                          CommonGrid,varargin{3},varargin{4});
0060   else
0061       CommonGrid = varargin{1};
0062       UCommon = varargin{2};
0063   end
0064
0065 %*************************** MAIN ***************************************
0066
0067   %Initialization
0068   nze = find(CommonGrid.N4E(:,1));
0069   MeanT = (Told+T)/2;
0070   EtaTemporal2 = 0;
0071   QWeights = Options.QWeights;
0072
0073   %obtain vertices
0074   A = CommonGrid.C4N(CommonGrid.N4E(nze,1),:);
0075   B = CommonGrid.C4N(CommonGrid.N4E(nze,2),:);
0076   C = CommonGrid.C4N(CommonGrid.N4E(nze,3),:);
0077   if isempty(CommonGrid.DoubleArea)
0078       %Assembly of areas
0079       CommonGrid.DoubleArea = ...
0080                       B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
0081                       -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0082   end
0083
0084   %gradients
0085   N1 = [B(:,2)-C(:,2) C(:,1)-B(:,1)]./(CommonGrid.DoubleArea*ones(1,2));
0086   N2 = [C(:,2)-A(:,2) A(:,1)-C(:,1)]./(CommonGrid.DoubleArea*ones(1,2));
0087   N3 = [A(:,2)-B(:,2) B(:,1)-A(:,1)]./(CommonGrid.DoubleArea*ones(1,2));
0088
0089   %Common U values
0090   U1 = UCommon(CommonGrid.N4E(nze,1),1);
0091   U2 = UCommon(CommonGrid.N4E(nze,2),1);
0092   U3 = UCommon(CommonGrid.N4E(nze,3),1);
0093   U1old = UCommon(CommonGrid.N4E(nze,1),2);
0094   U2old = UCommon(CommonGrid.N4E(nze,2),2);
0095   U3old = UCommon(CommonGrid.N4E(nze,3),2);
0096
0097   %Assembly of gradient term of energy norm
0098   a = ((U1-U1old)*ones(1,2)).*N1 ...
0099        + ((U2-U2old)*ones(1,2)).*N2 + ((U3-U3old)*ones(1,2)).*N3;
0100   sum1 = sum((a(:,1).^2+a(:,2).^2).*CommonGrid.DoubleArea)/2;
0101
0102   %Assembly of L2 term if energy norm
```

```
0103   sum2 = 0;
0104   for i=1:size(QWeights,1)
0105      sum2 = sum2 + QWeights(i)*...
0106             sum(CommonGrid.DoubleArea'.*(Options.QBasisfct(i,:)*...
0107                                 [(U1-U1old) (U2-U2old) (U3-U3old)]').^2);
0108   end
0109
0110   %Consider space-time dependent f-data
0111   if Options.Dependence.F==3
0112       for i=1:size(QWeights,1)
0113           QuadraturVertices = ...
0114                    A + (B-A)*Options.QNodes(i,1)+(C-A)*Options.QNodes(i,2);
0115           EtaTemporal2 = EtaTemporal2 + ...
0116                  QWeights(i)*sum(CommonGrid.DoubleArea.*...
0117                   (feval(Driver,'df(x,t)/dt',QuadraturVertices,MeanT)).^2);
0118       end
0119
0120   elseif Options.Dependence.F==1
0121
0122       %spatially constant rhs
0123       EtaTemporal2 = sum(CommonGrid.DoubleArea(CommonElement))/2*...
0124                                  (feval(Driver,'df(x,t)/dt',[0 0],MeanT))^2;
0125   end
0126
0127   %Check whether time dependent data
0128   if Options.Dependence.D==1 || Options.Dependence.D==3 ...
0129       || Options.Dependence.C==1 || Options.Dependence.C==3 ...
0130       || Options.Dependence.R==1 || Options.Dependence.R==3
0131        TimeDependentData = 1;
0132        dDdt = feval(Driver,'dD(x,t)/dt',[0 0],MeanT);
0133        dCdt = feval(Driver,'dc(x,t)/dt',[0 0],MeanT);
0134        dRdt = feval(Driver,'dr(x,t)/dt',[0 0],MeanT);
0135   else
0136        TimeDependentData = 0;
0137   end
0138
0139   %Consider time dependent data
0140   if TimeDependentData
0141
0142          MeanU1 = (U1+U1old)/2;
0143          MeanU2 = (U2+U2old)/2;
0144          MeanU3 = (U3+U3old)/2;
0145          MeanNablaU = (MeanU1*ones(1,2)).*N1 + ...
0146                       (MeanU2*ones(1,2)).*N2 + ...
0147                       (MeanU3*ones(1,2)).*N3;
0148
0149          if Options.Dependence.D==1
0150              a = dDdt*MeanNablaU';
0151              EtaTemporal2 = EtaTemporal2 +...textcolorcomment
```

```
0152          Lambda^(-1)/2*sum((a(1,:).^2+a(2,:).^2).*CommonGrid.DoubleArea');
0153          elseif Options.Dependence.D==3
0154              for j=1:size(QWeights,1)
0155                QuadraturVertices = ...
0156                    A + (B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0157                dDdt = feval(Driver,'dD(x,t)/dt',QuadraturVertices,MeanT);
0158                dDdt = reshape([dDdt(1:size(A,1),1)';...
0159                                dDdt(size(A,1)+1:end,1)';...
0160                                dDdt(1:size(A,1),2)';...
0161                                dDdt(size(A,1)+1:end,2)'],2,2*size(A,1))';
0162                dDdt = blktridiag(dDdt,size(A,1));
0163                a = reshape(dDdt*...
0164                          reshape(MeanNablaU',2*size(A,1),1),2,size(A,1))';
0165                a = a(:,1).^2 + a(:,2).^2;
0166                EtaTemporal2 = EtaTemporal2...
0167                    + Lambda^(-1)*QWeights(j)*sum(a.*CommonGrid.DoubleArea);
0168             end
0169          end
0170          if Options.Dependence.C==1
0171              a = dCdt*MeanNablaU';
0172              EtaTemporal2 = EtaTemporal2 + ...
0173                                          sum(a'.^2.*CommonGrid.DoubleArea)/2;
0174          elseif Options.Dependence.C==3
0175              for j=1:size(QWeights,1)
0176                  QuadraturVertices = ...
0177                    A + (B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0178                  dCdt = feval(Driver,'dc(x,t)/dt',QuadraturVertices,MeanT);
0179                  a = dot(dCdt,MeanNablaU,2);
0180                  EtaTemporal2 = EtaTemporal2 +...
0181                              QWeights(j)*sum(a.^2.*CommonGrid.DoubleArea);
0182              end
0183          end
0184          if Options.Dependence.R==1
0185              for j=1:size(QWeights,1)
0186                  a = (Options.QBasisfct(j,:)*[MeanU1 MeanU2 MeanU3]').^2;
0187                  EtaTemporal2 = EtaTemporal2 +...
0188                         QWeights(j)*dRdt*sum(a'.*CommonGrid.DoubleArea).^2;
0189              end
0190          elseif Options.Dependence.R==3
0191              for j=1:size(QWeights,1)
0192                  QuadraturVertices = ...
0193                    A + (B-A)*Options.QNodes(j,1)+(C-A)*Options.QNodes(j,2);
0194                  dRdt = feval(Driver,'dr(x,t)/dt',QuadraturVertices,MeanT);
0195                  a = (Options.QBasisfct(j,:)*[MeanU1 MeanU2 MeanU3]').^2;
0196                  EtaTemporal2 = EtaTemporal2 +...
0197                        QWeights(j)*sum(dRdt.*a'.*CommonGrid.DoubleArea).^2;
0198              end
0199          end
0200      end
```

```
0201
0202   %Consider time dependent Neumann data
0203   if Options.Dependence.G
0204       switch Options.Dependence.G
0205           case 1
0206                LengthNeumannBoundary = 0;
0207                for i=1:size(CommonGrid.N4N,1)
0208                    LengthNeumannBoundary = LengthNeumannBoundary ...
0209                            + norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)-...
0210                                    CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0211                end
0212                EtaTemporal2 = EtaTemporal2 + LengthNeumannBoundary *...
0213                                (feval(Driver,'dg(x,t)/dt',[0 0],MeanT))^2;
0214           case 3
0215                for i=1:size(CommonGrid.N4N,1)
0216                    QuadraturVertices = ...
0217                                 (CommonGrid.C4N(CommonGrid.N4N(i,1),:)+...
0218                                  CommonGrid.C4N(CommonGrid.N4N(i,2),:))/2;
0219                    Length = norm(CommonGrid.C4N(CommonGrid.N4N(i,1),:)-...
0220                                    CommonGrid.C4N(CommonGrid.N4N(i,2),:));
0221                    EtaTemporal2 = EtaTemporal2 + Length *...
0222                     (feval(Driver,'dg(x,t)/dt',QuadraturVertices,MeanT))^2;
0223                end
0224
0225       end
0226   end
0227
0228   %Norm |||Un_1-Un|||
0229   EtaTemporal.Part1 = sqrt(Lambda*sum1 + Beta*sum2);
0230
0231   EtaTemporal.Part2 = sqrt(EtaTemporal2);
0232
0233   EtaTemporal.TemporalErrorEstimator = (EtaTemporal.Part1^2 ...
0234               + (1+Options.Theta)^2*(T-Told)^2*EtaTemporal.Part2^2)^(1/2);
0235
0236   %Finalize logging
0237   if Options.Logging
0238       LogStruct.ComputationTime = cputime - CPU_time;
0239   end
0240
0241   if Options.ShowTime
0242       ComputationTimeEtaTemporal = cputime - CPU_time
0243   end
0244
0245   %end of temporalError
0246 %-------------------------------------------------------------------------
0247 function UCommon = getCommonU(OldGrid,Grid,CommonGrid,Uold,U)
0248 %getCommonU:obtaining values U,Uold at the vertices of CommonGrid by linear
0249 %interpolation
```

```
0250
0251      nzec = find(CommonGrid.N4E(:,1));
0252      A = CommonGrid.C4N(CommonGrid.N4E(nzec,1),:);
0253      B = CommonGrid.C4N(CommonGrid.N4E(nzec,2),:);
0254      C = CommonGrid.C4N(CommonGrid.N4E(nzec,3),:);
0255
0256      Nodes = Grid.N4E(CommonGrid.N4E(nzec,4),1:3);
0257      OldNodes = OldGrid.N4E(CommonGrid.N4E(nzec,5),1:3);
0258      NewA = Grid.C4N(Nodes(:,1),:);
0259      NewB = Grid.C4N(Nodes(:,2),:);
0260      NewC = Grid.C4N(Nodes(:,3),:);
0261      OldA = OldGrid.C4N(OldNodes(:,1),:);
0262      OldB = OldGrid.C4N(OldNodes(:,2),:);
0263      OldC = OldGrid.C4N(OldNodes(:,3),:);
0264      U1 = U(Nodes(:,1));
0265      U2 = U(Nodes(:,2));
0266      U3 = U(Nodes(:,3));
0267      U1old = Uold(OldNodes(:,1));
0268      U2old = Uold(OldNodes(:,2));
0269      U3old = Uold(OldNodes(:,3));
0270      UCommon = zeros(size(CommonGrid.C4N,1),2);
0271
0272      %Obtain values of U on new grid
0273      M1 = reshape([NewB-NewA NewC-NewA]',2,2*size(NewA,1))';
0274      M1 = blktridiag(M1,size(NewA,1));
0275      CNodesRef = M1\reshape((A - NewA)',2*size(NewA,1),1);
0276      CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
0277      UCommon(CommonGrid.N4E(nzec,1),2)= ...
0278                  (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
0279                                      + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
0280      CNodesRef = M1\reshape((B - NewA)',2*size(NewA,1),1);
0281      CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
0282      UCommon(CommonGrid.N4E(nzec,2),2)= ...
0283                  (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
0284                                      + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
0285      CNodesRef = M1\reshape((C - NewA)',2*size(NewA,1),1);
0286      CNodesRef = reshape(CNodesRef,2,size(NewA,1))';
0287      UCommon(CommonGrid.N4E(nzec,3),2)= ...
0288                  (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1...
0289                                      + CNodesRef(:,1).*U2 + CNodesRef(:,2).*U3;
0290      %Obtain values of U on old grid
0291      M2 = reshape([OldB-OldA OldC-OldA]',2,2*size(OldA,1))';
0292      M2 = blktridiag(M2,size(OldA,1));
0293      CNodesRef = M2\reshape((A - OldA)',2*size(OldA,1),1);
0294      CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
0295      UCommon(CommonGrid.N4E(nzec,1),1)= ...
0296               (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
0297                                  + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
0298      CNodesRef = M2\reshape((B - OldA)',2*size(OldA,1),1);
```

```
0299     CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
0300     UCommon(CommonGrid.N4E(nzec,2),1)= ...
0301              (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
0302                           + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
0303     CNodesRef = M2\reshape((C - OldA)',2*size(OldA,1),1);
0304     CNodesRef = reshape(CNodesRef,2,size(OldA,1))';
0305     UCommon(CommonGrid.N4E(nzec,3),1)= ...
0306              (ones(size(NewA,1),1)-CNodesRef(:,1)-CNodesRef(:,2)).*U1old...
0307                           + CNodesRef(:,1).*U2old + CNodesRef(:,2).*U3old;
0308
0309     %end of getCommonU
0310
0311 %--------------------------------------------------------------------------
0312 function A = blktridiag(A,n)
0313 % BLKTRIDIAG: computes a sparse (block) tridiagonal matrix with n blocks
0314
0315 [q,p] = size(A);
0316 [ind1,ind2,ind3]=ndgrid(0:p-1,0:p-1,0:n-1);
0317 rind = 1+ind1(:)+p*ind3(:);
0318 cind = 1+ind2(:)+p*ind3(:);
0319 v = reshape(A',n*4,1);
0320 A = sparse(rind,cind,v,n*p,n*p);
0321
0322   %end of blktridiag
```

A.6 The Function *commonGrid*

```
0001 function [CommonGrid,GridsAreEqual,LogStruct] = ...                    %ok
0002                                                 commonGrid(OldGrid,Grid,Driver)
0003 %  commonGridcreates a common grid which is a refinement of OldGrid and
0004 %  Grid. Starting from Grid, CommonGrid is a further refinement thereof.
0005 %  Additionally, two columns(4,5) in N4E are introduced where the element
0006 %  in which the refined elements of CommonGrid are contained are saved
0007 %  (Grid(column 4),OldGrid (column 5) )
0008 %
0009 %   INPUT ARGUMENTS
0010 %
0011 %   Grid,OldGrid: structures with fields C4N,N4E,N4D,N4N,F4E,NOE,Unused,
0012 %                                       M,H,b,DoubleArea
0013 %
0014 %   [C4N: Coordinates;a nx2-matrix, where n is the number of vertices
0015 %       in the grid.
0016 %       Format of ith row: 1st coordinate of ith vertex/2nd coordinate of
0017 %       ith vertex
0018 %
0019 %   [N4E: Elements;nx5-matrix, where n is the number of triangles in
0020 %       the grid.F
0021 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0022 %       triangle/3rd vertex of ith triangle/father(referring to F4E-Matrix)
```

```
0023 %       /green brother]
0024 %
0025 %   [N4D: Dirichlet nodes; nx2-matrix, where n is the number of
0026 %       dirichlet edges.
0027 %       Format of ith row: 1st vertex of ith dirichlet edge/2nd vertex of
0028 %       ith dirichlet edge]
0029 %
0030 %   [N4N: Neumann nodes; see Dirichlet]
0031 %
0032 %   [F4E: Fathers; nx8-matrix, where n is the number of fathers in
0033 %       the grid. A father is a triangle which is split in 4 congruent
0034 %       subtriangles(called children) given by bisecting all edges of the
0035 %       father. A father is not in the actual Element-matrix, since it is
0036 %       not an actual element. His children may, but don't have to be in
0037 %       this matrix, they too can be splitted and may be in the
0038 %       Fathers-matrix;
0039 %       Format of ith row: 1st vertex of ith father/2nd vertex of ith
0040 %       father/3rd vertex of ith father/middle child/2nd child/3rd child
0041 %       /4th child/father of father
0042 %       Remark: column 1-3 referring to C4N-matrix
0043 %               column 4-7 referring to N4E-matrix
0044 %               column 8 referring to F4E-matrix ]
0045 %
0046 %   [NON: Nodes on Edges; nxn-matrix, where n is number of vertices.
0047 %       With NON the algorithm administers vertices (normal and
0048 %       hanging ones). In ith row/jth column there is the number of the
0049 %       vertex on the edge between the ith and jth vertice(referring to
0050 %       C4N-matrix) if there is none or node lies on a boundary edge
0051 %       the entry is 0]
0052 %
0053 %   [Unused.C4N/N4E/F4E: Unused rows from these matrices;
0054 %       Format: vertical vector with row number entries
0055 %
0056 %   [M,H,b,DoubleArea: Saved mass-matrix,
0057 %       transport-matrix, rhs and double areas for Grid. These fields are
0058 %       deleted, if a refinement process is conducted
0059 %
0060 %   [Driver: name of the file, where the problem data is located; string]
0061 %
0062 %   OUTPUT ARGUMENTS:
0063 %
0064 %   CommonGrid: structure with fields C4N,N4E,N4D,N4N,Unused,M,H,b,
0065 %                                     DoubleArea
0066 %   [C4N: Coordinates for the vertices of CommonGrid, see input]
0067 %
0068 %   [N4E: Elements; nx5-matrix, where n is the number of triangles in
0069 %       CommonGrid.
0070 %       Format of ith row: 1st vertex of ith triangle/2nd vertex of ith
0071 %       triangle/3rd vertex of ith triangle/element of Grid in which the
```

```
0072 %       ith element of CommonGrid is contained(referring to
0073 %       Grid.N4E-Matrix)/element of OldGrid in which the ith element of
0074 %       CommonGrid is contained(referring to OldGrid.N4E-Matrix)]
0075 %
0076 %   [GridsAreEqual: 1 if Grid and OldGrid are the same, 0 otherwise]
0077 %
0078 %   [LogStruct is a log-struct which contains information about the
0079 %    solution process.]
0080 %
0081 %
0082 %   See also parabolicSolver
0083 %
0084 %   Copyright (c) 2007 Philipp Wissgott
0085 %                      Vienna University of Technology
0086
0087 %************************** LOGGING ************************************
0088
0089   CPU_time = cputime;
0090   if nargout==3
0091       Options.Logging = 1;
0092   else
0093       Options.Logging = 0;
0094   end
0095
0096 %************************** OPTIONS ************************************
0097
0098   OptionFile = feval(Driver,'OptionFile');
0099   Options = feval(OptionFile,'LoadOptions','CommonGrid',Driver,0,1);
0100
0101 %************************** MAIN ***************************************
0102
0103
0104   %Check if Grid and OldGrid are the same
0105   nze = setdiff(1:size(Grid.N4E,1),Grid.Unused.N4E);
0106   if isequal(OldGrid.C4N,Grid.C4N) && isequal(OldGrid.N4E,Grid.N4E)
0107       GridsAreEqual = 1;
0108       CommonGrid = Grid;
0109       CommonGrid.N4E(nze,4:5) = [nze' nze'];
0110       return;
0111   else
0112       GridsAreEqual = 0;
0113   end
0114
0115   %Initialisation
0116
0117   %default value of CommonGrid is Grid
0118   CommonGrid = Grid;
0119   %prelocation
0120   Ngb = nnz(find(Grid.N4E(:,5))); %number of green brothers
```

```
0121  %all other elements
0122  Nngb = max(size(Grid.N4E,1)-nnz(Grid.Unused.N4E)- Ngb,0);
0123  CommonGrid.ElementList = zeros(Ngb*2+Nngb*3,5);
0124  CommonGrid.ElementCount = 0;
0125  CommonGrid.VertexCount = size(CommonGrid.C4N,1);
0126  CommonGrid.C4N = [CommonGrid.C4N; zeros(Ngb*2+Nngb*3,2)];
0127
0128  CommonGrid.N4E = [CommonGrid.N4E(:,1:3) zeros(size(CommonGrid.N4E,1),2)];
0129  a = CommonGrid.N4E(nze,1:3)';
0130  I1 = reshape(a,3*nnz(nze),1);
0131  a = CommonGrid.N4E(nze,[2 3 1])';
0132  I2 = reshape(a,3*nnz(nze),1);
0133  CommonGrid.NOE = CommonGrid.NOE ...
0134          + sparse(I1,I2,-1,size(CommonGrid.NOE,1),size(CommonGrid.NOE,1));
0135  CommonGrid.NOE = [CommonGrid.NOE sparse(size(CommonGrid.NOE,1),nnz(I1));
0136                    sparse(nnz(I1),size(CommonGrid.NOE,1)+nnz(I1))];
0137
0138  %find some of the unchanged elements
0139  Cand = setdiff(nze,size(OldGrid.N4E,1)+1:max(nze));
0140  a = Grid.N4E(Cand,1:3)-OldGrid.N4E(Cand,1:3);
0141  a = find(a(:,1).^2 + a(:,2).^2 + a(:,3).^2==0);
0142  if ~isempty(a)
0143
0144      A = Grid.C4N(Grid.N4E(Cand(a),1),:)...
0145                                      -OldGrid.C4N(OldGrid.N4E(Cand(a),1),:);
0146      B = Grid.C4N(Grid.N4E(Cand(a),2),:)...
0147                                      -OldGrid.C4N(OldGrid.N4E(Cand(a),2),:);
0148      C = Grid.C4N(Grid.N4E(Cand(a),3),:)...
0149                                      -OldGrid.C4N(OldGrid.N4E(Cand(a),3),:);
0150      b = find(A(:,1).^2+A(:,2).^2+B(:,1).^2 ...
0151                                          +B(:,2).^2+C(:,1).^2+C(:,2).^2==0);
0152      Unchanged = Cand(a(b));
0153      CommonGrid.N4E(Unchanged,4:5) = [Unchanged' Unchanged'];
0154      nze = setdiff(nze,Unchanged);
0155  end
0156
0157  %Delete saved data from CommonGrid
0158  CommonGrid.M = [];
0159
0160  for j = 1:size(nze,2)
0161
0162     Element = nze(j);
0163     %Collecting information about Element/stored in Grid
0164     Vertices = sortrowsP(Grid.C4N(Grid.N4E(Element,1:3),:));
0165     ElementGreenBrother = Grid.N4E(Element,5);
0166
0167     %default
0168     OldElement = Element;
0169     OldFather = 0;
```

```
0170
0171      %Check whether Element is a green element.
0172      if ElementGreenBrother
0173         %Assume that ElementGreenBrother comes first in Grid.N4E if Element
0174         %is lastborn
0175         if ElementGreenBrother>Element
0176            %default value
0177            ElementIsFirstBorn = 1;
0178            TwinVertices = sortrowsP([Grid.C4N(Grid.N4E(Element,2:3),:);...
0179                     Grid.C4N(Grid.N4E(ElementGreenBrother,2),:)]);
0180         else
0181            OldElement = ElementGreenBrother;%default2
0182            ElementIsFirstBorn = 0;
0183            TwinVertices = sortrowsP([Grid.C4N(Grid.N4E(Element,2:3),:);...
0184                     Grid.C4N(Grid.N4E(ElementGreenBrother,3),:)]);
0185         end
0186      end
0187
0188      Father = Grid.N4E(Element,4);
0189      if Father
0190         ElementNr = find(Grid.F4E(Father,4:7) == Element);
0191         if ElementGreenBrother && ~ElementIsFirstBorn
0192            ElementNr = find(Grid.F4E(Father,4:7) == ElementGreenBrother);
0193         end
0194      else
0195         ElementNr = 1;
0196      end
0197
0198      %Identification of OldElement/information stored in OldGrid
0199      %Finding right OldElement to relate Element to
0200
0201      if Father %check if initial element
0202         RelatedFathers = Father;
0203         OldElementFound = 0;
0204
0205         try
0206            %cases handled:4a,5,6a,7; Element may not exist in
0207            %OldGrid->catch;
0208            if ElementGreenBrother && ~ElementIsFirstBorn
0209                ElementTest = ElementGreenBrother;
0210            else
0211                ElementTest = Element;
0212            end
0213            OldVertices = ...
0214                     sortrowsP(OldGrid.C4N(OldGrid.N4E(ElementTest,1:3),:));
0215            OldFather = OldGrid.N4E(ElementTest,4);
0216            OldElementGreenBrother = OldGrid.N4E(ElementTest,5);
0217            OldElement1 = min(ElementTest,OldElementGreenBrother);
0218            OldElement2 = max(ElementTest,OldElementGreenBrother);
```

```
0219            OldFatherVertices = ...
0220                    sortrowsP(OldGrid.C4N(OldGrid.F4E(OldFather,1:3),:));
0221 if isequal(OldFatherVertices,Vertices)  ...  %Case5
0222    || (OldElementGreenBrother ...
0223        && isequal(sortrowsP([OldGrid.C4N(OldGrid.N4E(OldElement1,2:3),:);
0224           OldGrid.C4N(OldGrid.N4E(OldElement2,2),:)]),Vertices)) ...%Case4a
0225   || (OldElementGreenBrother && ElementGreenBrother && ...
0226       isequal(sortrowsP([OldGrid.C4N(OldGrid.N4E(OldElement1,2:3),:);
0227          OldGrid.C4N(OldGrid.N4E(OldElement2,2),:)]),...
0228          TwinVertices)) ...%Case6a
0229   || (ElementGreenBrother && isequal(OldFatherVertices,TwinVertices))%Case7
0230                 OldElementFound = 1;
0231           %OldElement remains at the default value Element/ElementFirstBorn
0232            end
0233         catch
0234         end
0235         if ~OldElementFound
0236              try
0237                  %Case handled 1,2,4b,6b;
0238                  MiddleElement = Grid.F4E(Father,4);
0239                  %trying to change to OldGrid
0240                  OldFather = OldGrid.N4E(MiddleElement,4);
0241                  OldFatherVertices = ...
0242                       sortrowsP(OldGrid.C4N(OldGrid.F4E(OldFather,1:3),:));
0243                  for i=1:4
0244                       OldElementTest = OldGrid.F4E(OldFather,3+i);
0245                       OldVertices = ...
0246                  sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElementTest,1:3),:));
0247                     OldElementGreenBrother = OldGrid.N4E(OldElementTest,5);
0248  if isequal(OldVertices,Vertices) ... %Case 1
0249     || (ElementGreenBrother && isequal(OldVertices,TwinVertices))...%Case2
0250     || (OldElementGreenBrother ... %Case4b
0251         && isequal(sortrowsP([OldGrid.C4N(OldGrid.N4E(OldElementTest,2:3),:);
0252       OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)]),Vertices)) ...
0253     || (OldElementGreenBrother && ElementGreenBrother && ... %Case6b
0254         isequal(sortrowsP([OldGrid.C4N(OldGrid.N4E(OldElementTest,2:3),:);
0255       OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)]),TwinVertices))
0256                           OldElement = OldElementTest;
0257                           OldElementFound = 1;
0258                           break;
0259                      end
0260                  end
0261              catch
0262              end
0263         end
0264         while OldElementFound == 0
0265             %Cases handled 3,8;
0266             RelatedFatherVertices = ...
0267                     sortrowsP(Grid.C4N(Grid.F4E(RelatedFathers(1),1:3),:));
```

```
0268            MiddleElement = Grid.F4E(RelatedFathers(1),4);
0269            %cases 3a,8a
0270            try
0271                OldFather = OldGrid.N4E(MiddleElement,4);
0272                for i=1:4
0273                    ElementTest = OldGrid.F4E(OldFather,3+i);
0274                    OldVertices = ...
0275                    sortrowsP(OldGrid.C4N(OldGrid.N4E(ElementTest,1:3),:));
0276                    OldElementGreenBrother = OldGrid.N4E(ElementTest,5);
0277                    RelatedFatherVertices2 = ...
0278                    sortrowsP(Grid.C4N(Grid.F4E(RelatedFathers(2),1:3),:));
0279                    if isequal(OldVertices,RelatedFatherVertices2)%Case 3a
0280                        OldElement = ElementTest;
0281                        OldFather = OldGrid.N4E(OldElement,4);
0282                        OldElementFound = 2;
0283                        break;
0284                    elseif OldElementGreenBrother
0285                        OldElementGreenBrotherVertices = ...
0286          sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,1:3),:));
0287                        OldTwinVertices = ...
0288        sortrowsP([OldGrid.C4N(OldGrid.N4E(ElementTest,2:3),:);...
0289        OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)]);
0290                        if isequal(OldTwinVertices,RelatedFatherVertices2)
0291                            %Case 8a
0292                            OldElement = ...
0293                                min(ElementTest,OldElementGreenBrother);
0294                            OldVertices = ...
0295                     sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:));
0296                            OldFather = OldGrid.N4E(OldElement,4);
0297                            OldElementFound = 2;
0298                            break;
0299                        end
0300                    end
0301                end
0302                if OldElementFound
0303                 RelatedFathers = RelatedFathers(2:length(RelatedFathers));
0304                   break;
0305                end
0306            catch
0307            end
0308            %cases 3b,8b
0309            try
0310                %trying to change to OldGrid
0311                OldVertices = ...
0312                  sortrowsP(OldGrid.C4N(OldGrid.N4E(MiddleElement,1:3),:));
0313                OldElementGreenBrother = OldGrid.N4E(MiddleElement,5);
0314                if isequal(OldVertices,RelatedFatherVertices) %case 3b
0315                     OldElement = MiddleElement;
0316                     OldVertices = ...
```

```
0317                    sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:));
0318                    OldFather = OldGrid.N4E(OldElement,4);
0319                    OldElementFound = 2;
0320                elseif OldElementGreenBrother
0321                    OldElementGreenBrotherVertices = ...
0322         sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,1:3),:));
0323                    OldTwinVertices = ...
0324          sortrowsP([OldGrid.C4N(OldGrid.N4E(MiddleElement,2:3),:);...
0325          OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)]);
0326                    if isequal(OldTwinVertices,RelatedFatherVertices)%8b
0327                        OldElement = ...
0328                                min(MiddleElement,OldElementGreenBrother);
0329                        OldVertices = ...
0330                    sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:));
0331                        OldFather = OldGrid.N4E(OldElement,4);
0332                        OldElementFound = 2;
0333                    else
0334                        RelatedFathers = ...
0335                          [Grid.F4E(RelatedFathers(1),8); RelatedFathers];
0336                    end
0337                else
0338                    RelatedFathers = ...
0339                          [Grid.F4E(RelatedFathers(1),8); RelatedFathers];
0340                end
0341            catch
0342                RelatedFathers = ...
0343                          [Grid.F4E(RelatedFathers(1),8); RelatedFathers];
0344            end
0345
0346         end
0347      else %if Father
0348         %Element is a initial element
0349         %->OldElement=Element/ElementGreenBrother
0350         if ElementGreenBrother && ~ElementIsFirstBorn
0351            OldElement = ElementGreenBrother;
0352         else
0353            OldElement = Element;
0354         end
0355         OldVertices = ...
0356                    sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:));
0357         OldFather = OldGrid.N4E(OldElement,4);
0358         OldElementFound = 2;
0359      end %if Father
0360
0361      %end of identification
0362
0363      OldElementGreenBrother = OldGrid.N4E(OldElement,5);
0364      if OldElementGreenBrother
0365         OldElementGreenBrotherVertices = ...
```

```
0366          sortrowsP(OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,1:3),:));
0367        OldTwinVertices = ...
0368                      sortrowsP([OldGrid.C4N(OldGrid.N4E(min(OldElement,...
0369                                             OldElementGreenBrother),2:3),:);
0370             OldGrid.C4N(OldGrid.N4E(max(OldElement,...
0371                                             OldElementGreenBrother),2),:)]);
0372      end
0373      if OldFather && (OldElementFound == 2)
0374          OldFatherVertices = ...
0375                      sortrowsP(OldGrid.C4N(OldGrid.F4E(OldFather,1:3),:));
0376      end
0377
0378      %Set identifier
0379      NotIdentified = 1; %prevent multiple checks for performance
0380      if OldElementGreenBrother
0381         %default1: green->red(multiple)
0382         CommonIdentifier = 8;
0383      else
0384         %default2: white->red(multiple)
0385         CommonIdentifier = 3;
0386      end
0387
0388      if isequal(OldVertices,Vertices) || (OldElementGreenBrother  ...
0389                        && isequal(OldElementGreenBrotherVertices,Vertices))
0390          %no change
0391          CommonIdentifier = 1;
0392          NotIdentified = 0;
0393      end
0394
0395      if  NotIdentified && (ElementGreenBrother ...
0396            && ~OldElementGreenBrother && isequal(OldVertices,TwinVertices))
0397          %white->green
0398          CommonIdentifier = 2;
0399          NotIdentified = 0;
0400      end
0401
0402      if  NotIdentified && ~ElementGreenBrother ...
0403              && OldElementGreenBrother && isequal(OldTwinVertices,Vertices)
0404          %green->white
0405          CommonIdentifier = 4;
0406          NotIdentified = 0;
0407      end
0408
0409      if  NotIdentified && ...
0410                           (OldFather && isequal(OldFatherVertices,Vertices))
0411          %red(+green)->white
0412          CommonIdentifier = 5;
0413          NotIdentified = 0;
0414      end
```

```
0415
0416      if  NotIdentified && ElementGreenBrother ...
0417          && OldElementGreenBrother ...
0418          && isequal(OldTwinVertices,TwinVertices)...
0419          && ~isequal(OldTwinVertices,Vertices)
0420          %green->green(different shape)
0421          CommonIdentifier = 6;
0422          NotIdentified = 0;
0423      end
0424
0425      if  NotIdentified && OldFather && ElementGreenBrother && ...
0426             isequal(OldFatherVertices,TwinVertices)
0427           %Element was red(+green) and changed to green
0428           CommonIdentifier = 7;
0429      end
0430
0431      %Compute local contribution to CommonGrid
0432      switch (CommonIdentifier)
0433
0434           case 1
0435               %no change
0436               if OldElementGreenBrother && ...
0437                           isequal(OldElementGreenBrotherVertices,Vertices)
0438                  CommonGrid.N4E(Element,4:5) = ...
0439                                          [Element OldElementGreenBrother];
0440               else
0441                  CommonGrid.N4E(Element,4:5) = [Element OldElement];
0442               end
0443
0444           case 2
0445               %white->green
0446               CommonGrid.N4E(Element,4:5) = [Element OldElement];
0447
0448           case 3
0449               %white->red(multiple)
0450               CommonGrid.N4E(Element,4:5) = [Element OldElement];
0451
0452           case 4
0453               %green->white
0454               CommonGrid = ...
0455                         commonGrid4(Element,OldElement,OldGrid,CommonGrid);
0456
0457           case 5
0458               %red(+green)->white
0459               CommonGrid = ...
0460                         commonGrid5(Element,OldElement,OldGrid,CommonGrid);
0461
0462           case 6
0463               %green->green(different shape)
```

```
0464                CommonGrid = commonGrid6(Element,OldElement,...
0465                                         ElementGreenBrother,OldGrid,CommonGrid);
0466
0467            case 7
0468                %red(+green)->green
0469                CommonGrid = commonGrid7(Element,ElementGreenBrother,...
0470                                                 OldElement,OldGrid,CommonGrid);
0471
0472            case 8
0473                %green->red(multiple)
0474                OldTwinVertices =[OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:);
0475                       OldGrid.C4N(OldGrid.N4E(OldElementGreenBrother,2),:)];
0476                CommonGrid = commonGrid8(Element,1,RelatedFathers,...
0477                   OldElement,OldTwinVertices,[1 2],OldGrid,Grid,CommonGrid);
0478
0479        end
0480
0481    end
0482
0483    %Update N4E
0484    if CommonGrid.ElementCount
0485        Nue = nnz(CommonGrid.Unused.N4E);
0486        if CommonGrid.ElementCount>Nue
0487            CommonGrid.N4E(CommonGrid.Unused.N4E,:) = ...
0488                                                 CommonGrid.ElementList(1:Nue,:);
0489            CommonGrid.Unused.N4E = [];
0490            CommonGrid.N4E = [CommonGrid.N4E; ...
0491                    CommonGrid.ElementList(Nue+1:CommonGrid.ElementCount,:)];
0492        else
0493           CommonGrid.N4E(CommonGrid.Unused.N4E(1:CommonGrid.ElementCount)...
0494                   ,:) = CommonGrid.ElementList(1:CommonGrid.ElementCount,:);
0495           CommonGrid.Unused.N4E = ...
0496                        CommonGrid.Unused.N4E(CommonGrid.ElementCount+1:end);
0497        end
0498        CommonGrid.ElementList = [];
0499    end
0500
0501    %Update C4N
0502    CommonGrid.C4N = CommonGrid.C4N(1:CommonGrid.VertexCount,:);
0503    %Regularisation
0504    CommonGrid = regularizeCommonGrid(CommonGrid);
0505
0506    %Finalize logging
0507    LogStruct.ComputationTime = cputime - CPU_time;
0508
0509    if Options.DisplayOutput
0510      %graphical representation
0511      show(CommonGrid,zeros(size(CommonGrid.C4N,1),1),'Common grid',1)
0512    end
```

```
0513
0514   if Options.ShowTime
0515       ComputationTimeCommonGrid = LogStruct.ComputationTime
0516   end
0517
0518  %end of commonGrid
0519 %--------------------------------------------------------------------------
0520 function CommonGrid = commonGrid4(Element,OldElement,OldGrid,CommonGrid)
0521   %commonGrid: Element was green and changed to white
0522
0523   Nodes = CommonGrid.N4E(Element,1:3);
0524   Vertices = CommonGrid.C4N(Nodes,:);
0525   OldVertices = OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:);
0526   OldElementGreenBrother = OldGrid.N4E(OldElement,5);
0527
0528   for j=1:3
0529
0530       if nnz(ismemberP2((Vertices(j,:) + Vertices(mod(j,3)+1,:))/2,...
0531                                                          OldVertices)) == 1
0532           [FourthNode,FourthVertex,CommonGrid] = ...
0533                           getNOE(CommonGrid,[Nodes(j); Nodes(mod(j,3)+1)]);
0534           CommonGrid.NOE(Nodes(j),Nodes(mod(j,3)+1)) = FourthNode;
0535           CommonGrid.C4N(FourthNode,:) = FourthVertex;
0536           TopNode = setdiff([1 2 3], [j,mod(j,3)+1]);
0537           CommonGrid.N4E(Element,:) = ...
0538                    [FourthNode Nodes(TopNode) Nodes(j) Element OldElement];
0539           CommonGrid.ElementCount = CommonGrid.ElementCount + 1;
0540           CommonGrid.ElementList(CommonGrid.ElementCount,:) = ...
0541                           [FourthNode Nodes(mod(j,3)+1) ...
0542                            Nodes(TopNode) Element OldElementGreenBrother];
0543
0544       end
0545
0546   end
0547
0548  %end of commonGrid4
0549 %--------------------------------------------------------------------------
0550 function CommonGrid = commonGrid5(Element,OldElement,OldGrid,CommonGrid)
0551 %commonGrid5: Element was red(+green) and changed to white
0552
0553   Nodes = CommonGrid.N4E(Element,1:3);
0554   Vertices = CommonGrid.C4N(Nodes,:);
0555
0556   %Information about OldElement
0557   OldFather = OldGrid.N4E(OldElement,4);
0558   OldChildren = OldGrid.F4E(OldFather,4:7);
0559
0560   %Middle child has special attributes->treat first
0561
```

```
0562    InnerNodes = zeros(1,3);
0563    InnerVertices = zeros(3,2);
0564
0565    %Creating/Loading middle childs vertices in CommonGrid
0566    for j=1:3
0567
0568        [InnerNodes(j),InnerVertices(j,:),CommonGrid] = ...
0569                            getNOE(CommonGrid,[Nodes(j); Nodes(mod(j,3)+1)]);
0570        CommonGrid.NOE(Nodes(j),Nodes(mod(j,3)+1)) = InnerNodes(j);
0571        CommonGrid.C4N(InnerNodes(j),:) = InnerVertices(j,:);
0572
0573    end
0574
0575    CommonGrid.N4E(Element,:) = [InnerNodes Element OldChildren(1)];
0576
0577    %Check the other OldChildren
0578    NewElements = [];
0579    for j=2:4
0580
0581        OldChildVertices = OldGrid.C4N(OldGrid.N4E(OldChildren(j),1:3),:);
0582        OldChildGreenBrother = OldGrid.N4E(OldChildren(j),5);
0583        if OldChildGreenBrother
0584            OldGreenBrotherVertices = ...
0585                        OldGrid.C4N(OldGrid.N4E(OldChildGreenBrother,1:3),:);
0586            OldChildTwinVertices = ...
0587               union(OldChildVertices,OldGreenBrotherVertices,'rows');
0588            %Find the right node that are shared by
0589            %Element and OldChildren(j) or its green brother
0590            a = intersect(OldChildTwinVertices,Vertices,'rows');
0591            a = [1 1 1]'*a-Vertices;
0592            a = abs(a(:,1))+ abs(a(:,2));
0593            CurrentNode = Nodes(~a);
0594            %Find the right inner node that is shared
0595            %by OldChildGreenBrother and middle child
0596            a = [OldChildVertices(3,:);OldGreenBrotherVertices(2,:)];
0597            b = intersect(a,InnerVertices,'rows');
0598            b = [1 1 1]'*b-InnerVertices;
0599            b = abs(b(:,1))+ abs(b(:,2));
0600            InnerNode = InnerNodes(~b);
0601            %Find the bottom node that is shared
0602            %by the twins in CommonGrid
0603            [BottomNode,BottomVertex,CommonGrid] = ...
0604                                getNOE(CommonGrid,[CurrentNode; InnerNode]);
0605            CommonGrid.NOE(CurrentNode,InnerNode) = BottomNode;
0606            CommonGrid.C4N(BottomNode,:) = BottomVertex;
0607            %Find the top node that is shared by the twins in CommonGrid,
0608            %i.e. the vertex of the twins that has not been found so far
0609            TopNode = InnerNodes(mod(find(~b)+1,3)+1);
0610            %Assembly
```

```
0611            NewElements = [NewElements;...
0612                    [BottomNode TopNode CurrentNode Element OldChildren(j)]];
0613            NewElements = [NewElements; ...
0614                [BottomNode InnerNode TopNode Element OldChildGreenBrother]];
0615
0616        else
0617           %Find the right node that are shared by Element and OldChildren(j)
0618           %by fast 'ismember';
0619           I1 = ismemberP(Vertices,OldChildVertices);
0620           CurrentNode = Nodes(I1);
0621           %Find the right inner nodes that are shared by OldChildren(j) and
0622           %middle child
0623           I1 = ismemberP(InnerVertices,OldChildVertices);
0624           I2 = [true true true];
0625           I2(I1) = false;
0626           %Assembly
0627           NewElements = [NewElements; ...
0628               [CurrentNode InnerNodes(mod(find(I2)+1,3)+1) ...
0629                InnerNodes(mod(find(I2),3)+1) Element OldChildren(j)]];
0630        end
0631
0632    end
0633
0634    CommonGrid.ElementList(CommonGrid.ElementCount+1 ...
0635                :CommonGrid.ElementCount+size(NewElements,1),:) = NewElements;
0636    CommonGrid.ElementCount = CommonGrid.ElementCount + size(NewElements,1);
0637
0638    %end of commonGrid5
0639 %-------------------------------------------------------------------------
0640 function CommonGrid = commonGrid6(Element,OldElement,...
0641                                   ElementGreenBrother,OldGrid,CommonGrid)
0642 %commonGrid6: Element was green and changed to green (different shape)
0643
0644    Nodes = CommonGrid.N4E(Element,1:3);
0645
0646    %Information about OldElement
0647    OldVertices = OldGrid.C4N(OldGrid.N4E(OldElement,1:3),:);
0648    OldElementGreenBrother = OldGrid.N4E(OldElement,5);
0649
0650    %Assume that ElementGreenBrother comes first in Grid.N4E if Element is
0651    %lastborn
0652    if ElementGreenBrother>Element
0653       ElementIsFirstborn = 1;
0654       Order = [1 2 3];
0655    else
0656       Order = [1 3 2];
0657       ElementIsFirstborn = 0;
0658    end
0659
```

```
0660   [FourthNode,FourthVertex,CommonGrid] = getNOE(CommonGrid,Nodes(2:3));
0661   CommonGrid.NOE(Nodes(2),Nodes(3)) = FourthNode;
0662   CommonGrid.C4N(FourthNode,:) = FourthVertex;
0663   TopNode = Nodes(Order(2));
0664
0665   %new vertices are created/loaded: barycenter
0666   [BaryNode,BaryVertex,CommonGrid] = ...
0667                                       getNOE2(CommonGrid,[Nodes(1) TopNode]);
0668   CommonGrid.NOE(Nodes(1),TopNode) = BaryNode;
0669   CommonGrid.C4N(BaryNode,:) = BaryVertex;
0670   %Variable Affiliation considers whether a triangle is contained in
0671   %first or lastborn of OldElement
0672   Affiliation = zeros(3,2);
0673   a = [[BaryNode Nodes(Order(3)) Nodes(1)];
0674        [BaryNode FourthNode Nodes(Order(3))];
0675        [BaryNode TopNode FourthNode]];
0676   NewElements(:,1:3)= a(:,Order);
0677   %4 case selection due to non-symmetry-> orange refinement
0678   %(only three CommonGrid-elements are contained in Element)
0679   if ElementIsFirstborn
0680       if ~isempty(ismemberP2(OldVertices(1,:),FourthVertex))
0681           Affiliation(:,2) = ...
0682               [OldElementGreenBrother; OldElementGreenBrother; OldElement];
0683       else
0684           Affiliation(:,2) = ...
0685               [OldElement; OldElementGreenBrother; OldElementGreenBrother];
0686       end
0687   else
0688       if ~isempty(ismemberP2(OldVertices(1,:),FourthVertex))
0689          Affiliation(:,2) = ...
0690                              [OldElement; OldElement; OldElementGreenBrother];
0691       else
0692          Affiliation(:,2) = ...
0693                              [OldElementGreenBrother; OldElement; OldElement];
0694       end
0695   end
0696   Affiliation(:,1) = Element;
0697   NewElements(:,4:5) = Affiliation;
0698   CommonGrid.N4E(Element,:) = NewElements(1,:);
0699   CommonGrid.ElementList(CommonGrid.ElementCount+1 ...
0700                          :CommonGrid.ElementCount+2,:) = NewElements(2:3,:);
0701   CommonGrid.ElementCount = CommonGrid.ElementCount + 2;
0702
0703   %end of commonGrid6
0704 %-------------------------------------------------------------------------
0705 function CommonGrid = commonGrid7(Element,ElementGreenBrother,...
0706                                       OldElement,OldGrid,CommonGrid)
0707 %commonGrid7:Element was red(+green) and changed to green
0708
```

```
0709   Nodes = CommonGrid.N4E(Element,1:3);
0710   Vertices = CommonGrid.C4N(Nodes,:);
0711
0712   ElementGreenBrotherNodes = CommonGrid.N4E(ElementGreenBrother,1:3);
0713
0714   %Assume that ElementGreenBrother comes first in Grid.N4E if
0715   %Element is lastborn
0716   if ElementGreenBrother>Element
0717      ElementIsFirstBorn = 1;
0718      Order = [1 2 3];
0719   else
0720      Order = [1 3 2];
0721      ElementIsFirstBorn = 0;
0722   end
0723
0724
0725   TopVertex =  Vertices(Order(2),:);
0726   TopNode = Nodes(Order(2));
0727
0728   %Information about OldElement
0729   OldFather = OldGrid.N4E(OldElement,4);
0730   OldChildren = OldGrid.F4E(OldFather,4:7);
0731
0732   %Middle child has special attributes->treat first
0733   %The new "MiddleChild consists of 1.MiddleNode, the bisection of the
0734   %FourthNodes of Element and its GreenBrother ,2/3 FourthNode1 /Nodes(1)
0735   %Two nodes are not shared by Element but their edge is bisected by green
0736   %edge of Element
0737   if ElementIsFirstBorn
0738      [FourthNode1,FourthVertex1,CommonGrid] = ...
0739                                                 getNOE(CommonGrid,Nodes(2:3));
0740      CommonGrid.NOE(Nodes(2),Nodes(3)) = FourthNode1;
0741      CommonGrid.C4N(FourthNode1,:) = FourthVertex1;
0742      [FourthNode2,FourthVertex2,CommonGrid] = ...
0743                             getNOE(CommonGrid,ElementGreenBrotherNodes(2:3));
0744      CommonGrid.NOE(ElementGreenBrotherNodes(2),...
0745                                 ElementGreenBrotherNodes(3)) = FourthNode2;
0746      CommonGrid.C4N(FourthNode2,:) = FourthVertex2;
0747      [MiddleNode,MiddleVertex,CommonGrid] = ...
0748                               getNOE(CommonGrid,[FourthNode2; FourthNode1]);
0749      CommonGrid.NOE(FourthNode2,FourthNode1) = MiddleNode;
0750      CommonGrid.C4N(MiddleNode,:) = MiddleVertex;
0751      CommonGrid.N4E(Element,:) = ...
0752                    [MiddleNode FourthNode1 Nodes(1) Element OldChildren(1)];
0753   else
0754       MiddleNode = ElementGreenBrotherNodes(1);
0755       FourthNode1 = CommonGrid.NOE(Nodes(2),Nodes(3));
0756       FourthVertex1 = CommonGrid.C4N(CommonGrid.NOE(Nodes(2),Nodes(3)),:);
0757       CommonGrid.N4E(Element,:) = [MiddleNode Nodes(1) FourthNode1 ...
```

```
0758                                    Element OldChildren(1)];
0759    end
0760
0761    NewElements = [];
0762
0763    %Check the other OldChildren
0764    for j=2:4
0765
0766        OldChildVertices = OldGrid.C4N(OldGrid.N4E(OldChildren(j),1:3),:);
0767        OldChildGreenBrother = OldGrid.N4E(OldChildren(j),5);
0768        if OldChildGreenBrother
0769            OldChildTwinVertices = union(OldChildVertices,...
0770                OldGrid.C4N(OldGrid.N4E(OldChildGreenBrother,1:3),:),'rows');
0771        end
0772        %Check if top OldElement
0773        if ~isempty(ismemberP2(TopVertex,OldChildVertices)) ...
0774            || (OldChildGreenBrother  ...
0775            && ~isempty(ismemberP2(TopVertex,OldChildTwinVertices)))
0776           if OldChildGreenBrother
0777              %new vertices are created/loaded: 1.barycenter,
0778              %2.bisection of shared edge of Element and OldChild
0779              a = [TopNode FourthNode1];
0780              [FifthNode,FifthVertex,CommonGrid] = ...
0781                                      getNOE(CommonGrid,a(Order(2:3)-[1 1]));
0782              CommonGrid.NOE(a(Order(2)-1),a(Order(3)-1)) = FifthNode;
0783              CommonGrid.C4N(FifthNode,:) = FifthVertex;
0784              [BaryNode,BaryVertex,CommonGrid] = ...
0785                                     getNOE2(CommonGrid,[MiddleNode TopNode]);
0786              CommonGrid.NOE(MiddleNode,TopNode) = BaryNode;
0787              CommonGrid.C4N(BaryNode,:) = BaryVertex;
0788              %Variable Affiliation considers whether a triangle
0789              %is contained in first or lastborn of OldChild
0790              Affiliation = zeros(3,2);
0791              a = [[BaryNode FourthNode1 MiddleNode];
0792                   [BaryNode FifthNode FourthNode1];
0793                   [BaryNode TopNode FifthNode]];
0794               %4 case selection due to non-symmetry-> orange refinement
0795               %(only three CommonGrid-elements are contained in Element)
0796              if ElementIsFirstBorn
0797                 if ~isempty(ismemberP2(FifthVertex,OldChildVertices))
0798                    Affiliation(:,2) = ...
0799                [OldChildGreenBrother; OldChildGreenBrother; OldChildren(j)];
0800                 else
0801                    Affiliation(:,2) = ...
0802                [OldChildren(j); OldChildGreenBrother; OldChildGreenBrother];
0803                 end
0804              else
0805                 if ~isempty(ismemberP2(FifthVertex,OldChildVertices(1,:)))
0806                    Affiliation(:,2) = ...
```

```
0807                    [OldChildren(j); OldChildren(j); OldChildGreenBrother];
0808                else
0809                   Affiliation(:,2) = ...
0810                    [OldChildGreenBrother; OldChildren(j); OldChildren(j)];
0811                end
0812             end
0813             Affiliation(:,1) = Element;
0814             NewElements = [NewElements; [a(:,Order) Affiliation]];
0815          else
0816             a = [MiddleNode TopNode FourthNode1];
0817            NewElements = [NewElements; [a(Order) Element OldChildren(j) ]];
0818          end
0819       end
0820
0821       %Check if contained in Element
0822       if (size(intersect(OldChildVertices,...
0823           [FourthVertex1; Vertices(1,:)],'rows'),1) == 2) ...
0824  || (OldChildGreenBrother  && size(intersect(OldChildTwinVertices,...
0825                             [FourthVertex1; Vertices(1,:)],'rows'),1) == 2)
0826          if OldChildGreenBrother
0827              if isequal(OldChildVertices(1,:),...
0828                        (Vertices(1,:)+Vertices(Order(3),:))/2)
0829                 [FifthNode,FifthVertex,CommonGrid] = ...
0830                              getNOE(CommonGrid,[Nodes(1) Nodes(Order(3))]);
0831                 CommonGrid.NOE(Nodes(1),Nodes(Order(3))) = FifthNode;
0832                 CommonGrid.C4N(FifthNode,:) = FifthVertex;
0833                 NewElements = [NewElements; ...
0834    [FifthNode FourthNode1 Nodes(mod(Order(2),3)+1) Element OldChildren(j)];
0835    [FifthNode Nodes(Order(2)-1) FourthNode1 Element OldChildGreenBrother]];
0836              else
0837                 a = [FourthNode1 Nodes(Order(3))];
0838                 [FifthNode,FifthVertex,CommonGrid] = ...
0839                                       getNOE(CommonGrid,a(Order(2:3)-[1 1]));
0840                 CommonGrid.NOE(a(Order(2)-1),a(Order(3)-1)) = FifthNode;
0841                 CommonGrid.C4N(FifthNode,:) = FifthVertex;
0842                 NewElements = [NewElements; ...
0843           [FifthNode Nodes(1) a(Order(2)-1) Element OldChildren(j)];
0844           [FifthNode a(Order(3)-1) Nodes(1) Element OldChildGreenBrother]];
0845              end
0846          else
0847             if ElementIsFirstBorn
0848                NewElements = [NewElements; ...
0849                   [Nodes(1) FourthNode1 Nodes(3)  Element OldChildren(j)]];
0850             else
0851                NewElements = [NewElements; ...
0852                    [Nodes(1) Nodes(2) FourthNode1 Element OldChildren(j)]];
0853             end
0854          end
0855       end
```

```
0856
0857    end
0858
0859    CommonGrid.ElementList(CommonGrid.ElementCount+1 ...
0860                :CommonGrid.ElementCount+size(NewElements,1),:) = NewElements;
0861    CommonGrid.ElementCount = CommonGrid.ElementCount + size(NewElements,1);
0862    %end of commonGrid7
0863 %-------------------------------------------------------------------------
0864 function CommonGrid = commonGrid8(Element,RelatedFather,RelatedFathers,...
0865              OldElement,OldTwinVertices,Orientation,OldGrid,Grid,CommonGrid)
0866 %commonGrid8: Element was green and changed to red(+green)+multiple
0867
0868    %Information about Element
0869    Nodes = CommonGrid.N4E(Element,1:3);
0870    Vertices = CommonGrid.C4N(Nodes,:);
0871    Father = Grid.N4E(Element,4);
0872    ElementGreenBrother = Grid.N4E(Element,5);
0873    if ElementGreenBrother
0874        TwinVertices = ...
0875        union(Vertices,Grid.C4N(Grid.N4E(ElementGreenBrother,1:3),:),'rows');
0876        if ElementGreenBrother>Element
0877            ElementIsFirstBorn = 1;
0878            Order = [1 2 3];
0879        else
0880            ElementIsFirstBorn = 0;
0881            Order = [1 3 2];
0882        end
0883    end
0884
0885    %Information about OldElement
0886    OldElementGreenBrother = OldGrid.N4E(OldElement,5);
0887    a = [OldElement; OldElementGreenBrother];
0888    %Variable Affiliation1 considers the Orientation of OldTwin
0889    Affiliation1 = a(Orientation);
0890    NewElements = [];
0891    RelatedFathers(RelatedFather);
0892
0893    if Father == RelatedFathers(RelatedFather)
0894        if ~isempty(ismemberP2(OldTwinVertices(3,:),Vertices)) ...
0895                || (ElementGreenBrother ...
0896                  && ~isempty(ismemberP2(OldTwinVertices(3,:),TwinVertices)))
0897            %Element lies on the left side and is completely contained
0898            %in Affiliation1(1)
0899            CommonGrid.N4E(Element,4:5) = [Element Affiliation1(1)];
0900        elseif ~isempty(ismemberP2(OldTwinVertices(4,:),Vertices)) ...
0901                || (ElementGreenBrother ...
0902                && ~isempty(ismemberP2(OldTwinVertices(4,:),TwinVertices)))
0903            %Element lies on the right side and is completely contained
0904            %in Affiliation1(2)
```

```
0905            CommonGrid.N4E(Element,4:5) = [Element Affiliation1(2)];
0906        else
0907            if ~isempty(ismemberP2(OldTwinVertices(1,:),Vertices))...
0908               || (ElementGreenBrother ...
0909               && ~isempty(ismemberP2(OldTwinVertices(1,:),TwinVertices)))
0910                %Element lies in the middle and is split by the green edge
0911                if ElementGreenBrother
0912                    %Two important vertices of OldTwin
0913                    OldFourthVertex1 = ...
0914                              (OldTwinVertices(2,:)+OldTwinVertices(3,:))/2;
0915                    OldFourthVertex2 = ...
0916                              (OldTwinVertices(4,:)+OldTwinVertices(2,:))/2;
0917                    if ~isempty(ismemberP2((OldFourthVertex1+...
0918                                    OldFourthVertex2)/2,Vertices(1,:)))
0919                        if ElementIsFirstBorn
0920                           %Element is completely on the right side
0921                           CommonGrid.N4E(Element,4:5) = ...
0922                                                [Element Affiliation1(2)];
0923                        else
0924                           %Element is completely on the left side
0925                           CommonGrid.N4E(Element,4:5) = ...
0926                                                [Element Affiliation1(1)];
0927                        end
0928                    else
0929                       %Orange refinement necessary
0930                       %new vertices are created/loaded: 1.barycenter,
0931                       %2.bisection 2nd and 3rd node of Element
0932                       [BaryNode,BaryVertex,CommonGrid] = ...
0933                                      getNOE2(CommonGrid,Nodes(Order(1:2)));
0934                       CommonGrid.NOE(Nodes(Order(1)),Nodes(Order(2))) = ...
0935                                                              BaryNode;
0936                       CommonGrid.C4N(BaryNode,:) = BaryVertex;
0937                       [FifthNode,FifthVertex,CommonGrid] = ...
0938                                           getNOE(CommonGrid,Nodes(2:3));
0939                       CommonGrid.NOE(Nodes(2),Nodes(3)) = FifthNode;
0940                       CommonGrid.C4N(FifthNode,:) = FifthVertex;
0941                       %Variable Affiliation2 will save whether a triangle
0942                       %is contained in first or lastborn of OldTwin
0943                       Affiliation2 = zeros(3,2);
0944                       a = [[BaryNode Nodes(Order(3)) Nodes(1)];
0945                            [BaryNode FifthNode Nodes(Order(3))];
0946                            [BaryNode Nodes(Order(2)) FifthNode]];
0947                       NewElements(:,1:3)= a(:,Order);
0948                       %4 case selection due to non-symmetry->
0949                       %orange refinement(only three CommonGrid-elements
0950                       %are contained in Element)
0951                       if ElementIsFirstBorn
0952                          if ~isempty(ismemberP2(OldFourthVertex1,...
0953                                           Vertices(Order(2),:)))
```

```
0954                        Affiliation2(:,2) = ...
0955                      [Affiliation1(2); Affiliation1(1); Affiliation1(1)];
0956                       else
0957                        Affiliation2(:,2) = ...
0958                      [Affiliation1(1); Affiliation1(1); Affiliation1(2)];
0959                       end
0960                    else
0961                       if ~isempty(ismemberP2(OldFourthVertex1,...
0962                                              Vertices(Order(2),:)))
0963                        Affiliation2(:,2) = ...
0964                      [Affiliation1(2); Affiliation1(2); Affiliation1(1)];
0965                       else
0966                        Affiliation2(:,2) = ...
0967                      [Affiliation1(1); Affiliation1(2); Affiliation1(2)];
0968                       end
0969                    end
0970                    Affiliation2(:,1) = Element;
0971                    NewElements(:,4:5) = Affiliation2;
0972                end
0973             else
0974                a = Vertices - [1 1 1]'*OldTwinVertices(1,:);
0975                a = abs(a(:,1))+ abs(a(:,2));
0976                %BottomNode is the green node of Affiliation1(1)
0977                %in CommonGrid
0978                BottomNode = Nodes(~a);
0979                %MiddleNodes are the nodes of the middle child without
0980                %the green node of Affiliation1(1)
0981                MiddleNodes(1)= Nodes(mod(find(~a),3)+1);
0982                MiddleNodes(2)= Nodes(mod(find(~a)+1,3)+1);
0983                %creating node: MiddleNode lies on bisection of
0984                %MiddleNodes
0985                [MiddleNode,MiddleVertex,CommonGrid] = ...
0986                                          getNOE(CommonGrid,MiddleNodes);
0987                CommonGrid.NOE(MiddleNodes(1),MiddleNodes(2)) = ...
0988                                                          MiddleNode;
0989                CommonGrid.C4N(MiddleNode,:) = MiddleVertex;
0990                %A twin is created
0991                NewElements = ...
0992          [[MiddleNode BottomNode MiddleNodes(1) Element Affiliation1(2)];
0993           [MiddleNode MiddleNodes(2) BottomNode Element Affiliation1(1)]];
0994             end
0995           else
0996             %Element lies on top and is split by the green edge
0997             if ElementGreenBrother
0998                %Two important vertices of OldTwin
0999                OldFourthVertex1 = ...
1000                          (OldTwinVertices(2,:)+OldTwinVertices(3,:))/2;
1001                OldFourthVertex2 = ...textcolorcomment.
1002                          (OldTwinVertices(4,:)+OldTwinVertices(2,:))/2;
```

```
1003                    if ~isempty(ismemberP2((OldFourthVertex1...
1004                                           +OldFourthVertex2)/2,Vertices(1,:)))
1005                        if ElementIsFirstBorn
1006                           %Element is completely contained in
1007                           %Affiliation1(1)
1008                           CommonGrid.N4E(Element,4:5) = ...
1009                                                  [Element Affiliation1(1)];
1010                        else
1011                           %Element is completely contained in
1012                           %Affiliation1(2)
1013                           CommonGrid.N4E(Element,4:5) = ...
1014                                                  [Element Affiliation1(2)];
1015                        end
1016                    else
1017                       %Orange refinement necessary
1018                       %new vertices are created/loaded: 1.barycenter
1019                       %,2.bisection 2nd and 3rd node of Element
1020                       [BaryNode,BaryVertex,CommonGrid] = ...
1021                                        getNOE2(CommonGrid,Nodes(Order(1:2)));
1022                       CommonGrid.NOE(Nodes(Order(1)),Nodes(Order(2))) = ...
1023                                                                  BaryNode;
1024                       CommonGrid.C4N(BaryNode,:) = BaryVertex;
1025                       [FifthNode,FifthVertex,CommonGrid] = ...
1026                                                 getNOE(CommonGrid,Nodes(2:3));
1027                       CommonGrid.NOE(Nodes(2),Nodes(3)) = FifthNode;
1028                       CommonGrid.C4N(FifthNode,:) = FifthVertex;
1029                       %Variable Affiliation2 considers whether a triangle
1030                       %is contained in first or lastborn of Affiliation1(1)
1031                       Affiliation2 = zeros(3,2);
1032                       a = [[BaryNode Nodes(Order(3)) Nodes(1)];
1033                            [BaryNode FifthNode Nodes(Order(3))];
1034                            [BaryNode Nodes(Order(2)) FifthNode]];
1035                       NewElements(:,1:3)= a(:,Order);
1036                       %4 case selection due to non-symmetry->
1037                       %orange refinement (only three
1038                       %CommonGrid-elements are contained in Element)
1039                       if ElementIsFirstBorn
1040                          if ~isempty(ismemberP2(OldFourthVertex1,...
1041                                                 Vertices(Order(2),:)))
1042                             Affiliation2(:,2) = ...
1043                           [Affiliation1(2); Affiliation1(2); Affiliation1(1)];
1044                          else
1045                             Affiliation2(:,2) = ...
1046                           [Affiliation1(1); Affiliation1(2); Affiliation1(2)];
1047                          end
1048                       else
1049                          if ~isempty(ismemberP2(OldFourthVertex1,...
1050                                                 Vertices(Order(2),:)))
1051                             Affiliation2(:,2) = ...
```

```
1052                        [Affiliation1(2); Affiliation1(1); Affiliation1(1)];
1053                         else
1054                            Affiliation2(:,2) = ...
1055                        [Affiliation1(1); Affiliation1(1); Affiliation1(2)];
1056                         end
1057                      end
1058                      Affiliation2(:,1) = Element;
1059                      NewElements(:,4:5) = Affiliation2;
1060                  end
1061              else
1062                  a = [1 1 1]'*OldTwinVertices(2,:) - Vertices;
1063                  a = abs(a(:,1))+ abs(a(:,2));
1064                  %TopNode is the green node of Affiliation1(1)
1065                  %in CommonGrid
1066                  TopNode = Nodes(~a);
1067                  %MiddleNodes are the nodes of the middle child without
1068                  %the green node of OldChild
1069                  MiddleNodes(1)= Nodes(mod(find(~a),3)+1);
1070                  MiddleNodes(2)= Nodes(mod(find(~a)+1,3)+1);
1071                  %creating node: MiddleNode lies on bisection of
1072                  %MiddleNodes
1073                  [MiddleNode,MiddleVertex,CommonGrid] = ...
1074                                                  getNOE(CommonGrid,MiddleNodes);
1075                  CommonGrid.NOE(MiddleNodes(1),MiddleNodes(2)) = ...
1076                                                                  MiddleNode;
1077                  CommonGrid.C4N(MiddleNode,:) = MiddleVertex;
1078                  %A twin is created
1079                  NewElements = ...
1080             [[MiddleNode TopNode MiddleNodes(1) Element Affiliation1(1)];
1081              [MiddleNode MiddleNodes(2) TopNode Element Affiliation1(2)]];
1082              end
1083          end
1084      end
1085
1086  else
1087      RelatedFatherVertices = ...
1088                 Grid.C4N(Grid.F4E(RelatedFathers(RelatedFather+1),1:3),:);
1089      if size(intersect(OldTwinVertices(3,:),...
1090                                      RelatedFatherVertices,'rows'),1) == 1
1091          %Element lies on the left side and is completely contained
1092          %in Affiliation1(1)
1093           CommonGrid.N4E(Element,4:5) = [Element Affiliation1(1)];
1094      elseif size(intersect(OldTwinVertices(4,:),...
1095                                      RelatedFatherVertices,'rows'),1) == 1
1096          %Element lies on the right side and is completely contained
1097          %in Affiliation1(2)
1098          CommonGrid.N4E(Element,4:5) = [Element Affiliation1(2)];
1099      else
1100          if size(intersect(OldTwinVertices(1,:),...
```

```
1101                                    RelatedFatherVertices,'rows'),1) == 1
1102              %RelatedFather lies in the middle and is split by the
1103              %green edge->
1104              a = [1 1 1]'*OldTwinVertices(1,:) - RelatedFatherVertices;
1105              a = abs(a(:,1))+ abs(a(:,2));
1106              MiddleVertices(1,:) = ...textcolorcomment
1107                                    RelatedFatherVertices(mod(find(~a),3)+1,:);
1108              MiddleVertices(2,:) = ...
1109                                  RelatedFatherVertices(mod(find(~a)+1,3)+1,:);
1110              NewOldTwinVertices = ...
1111                                  [(MiddleVertices(1,:)+MiddleVertices(2,:))/2;
1112                                    OldTwinVertices(1,:);
1113                                    MiddleVertices(1,:);
1114                                    MiddleVertices(2,:)];
1115             %going one level down->recursive call
1116             CommonGrid = commonGrid8(Element,RelatedFather+1,...
1117                  RelatedFathers,OldElement,NewOldTwinVertices,...
1118                  [Orientation(2) Orientation(1)],OldGrid,Grid,CommonGrid);
1119          else
1120              %RelatedFather lies on top and is split by the green edge
1121              %RelatedFather lies in the middle and is split by the
1122              %green edge->
1123              a = [1 1 1]'*OldTwinVertices(2,:) - RelatedFatherVertices;
1124              a = abs(a(:,1))+ abs(a(:,2));
1125              MiddleVertices(1,:) = ...
1126                                    RelatedFatherVertices(mod(find(~a),3)+1,:);
1127              MiddleVertices(2,:) = ...
1128                                  RelatedFatherVertices(mod(find(~a)+1,3)+1,:);
1129              NewOldTwinVertices = ...
1130                                  [(MiddleVertices(1,:)+MiddleVertices(2,:))/2;
1131                                        OldTwinVertices(2,:);
1132                                        MiddleVertices(1,:);
1133                                        MiddleVertices(2,:)];
1134              %going one level down
1135              CommonGrid = commonGrid8(Element,RelatedFather+1,...
1136                  RelatedFathers,OldElement,NewOldTwinVertices,...
1137                  [Orientation(1) Orientation(2)],OldGrid,Grid,CommonGrid);
1138          end
1139      end
1140  end
1141  if ~isempty(NewElements)
1142    CommonGrid.N4E(Element,:) = NewElements(1,:);
1143    CommonGrid.ElementList(CommonGrid.ElementCount+1 ...
1144         :CommonGrid.ElementCount+size(NewElements,1)-1,:) = ...
1145                                     NewElements(2:size(NewElements,1),:);
1146    CommonGrid.ElementCount = CommonGrid.ElementCount ...
1147                                                 + size(NewElements,1) - 1;
1148  end
1149
```

```
1150   %end of commonGrid8
1151 %--------------------------------------------------------------------------
1152 function [Node,Vertex,Grid] = getNOE(Grid,Nodes)
1153 %getNOE: gets either an existing node at the bisection of Nodes(1)
1154 %and Nodes(2) or creates a new one
1155
1156   if Grid.NOE(Nodes(2),Nodes(1))>0
1157       %vertex already exists
1158       Node = Grid.NOE(Nodes(2),Nodes(1));
1159       Vertex = Grid.C4N(Node,:);
1160   else
1161       %new vertex is created
1162       Vertex = (Grid.C4N(Nodes(1),:)+Grid.C4N(Nodes(2),:))/2;
1163       if isempty(Grid.Unused.C4N)
1164          Grid.VertexCount = Grid.VertexCount + 1;
1165          Node = Grid.VertexCount;
1166       else
1167          Node = Grid.Unused.C4N(1);
1168          Grid.Unused.C4N = Grid.Unused.C4N(2:size(Grid.Unused.C4N,1));
1169       end
1170   end
1171
1172   %end of getNOE
1173 %--------------------------------------------------------------------------
1174 function [Node,Vertex,Grid] = getNOE2(Grid,Nodes)
1175 %gets either an existing node at 1/3 from
1176 %Nodes(1) to Nodes(2) or creates a new one
1177
1178   if Grid.NOE(Nodes(1),Nodes(2))>0
1179       %vertex already exists
1180       Node = Grid.NOE(Nodes(1),Nodes(2));
1181       Vertex = Grid.C4N(Node,:);
1182   else
1183       %new vertex is created
1184       Vertex = Grid.C4N(Nodes(1),:)+(Grid.C4N(Nodes(2),:)...
1185                -Grid.C4N(Nodes(1),:))/3;
1186       if isempty(Grid.Unused.C4N)
1187          Grid.VertexCount = Grid.VertexCount + 1;
1188          Node = Grid.VertexCount;
1189       else
1190          Node = Grid.Unused.C4N(1);
1191          Grid.Unused.C4N = Grid.Unused.C4N(2:size(Grid.Unused.C4N,1));
1192      end
1193   end
1194
1195   %end of getNOE2
1196
1197 %--------------------------------------------------------------------------
1198 function CommonGrid = regularizeCommonGrid(CommonGrid)
```

```
1199 %regularizeCommonGrid: updates edges,calls greenCompletion
1200
1201   %renaming for readability
1202   C4N = CommonGrid.C4N;
1203   N4E = CommonGrid.N4E;
1204   N4D = CommonGrid.N4D;
1205   N4N = CommonGrid.N4N;
1206   F4E = CommonGrid.F4E;
1207   NOE = CommonGrid.NOE;
1208
1209   %Deleting entries<0 of NOE
1210   NOE = (NOE + abs(NOE))/2;
1211
1212   %Update Dirichlet edges
1213   NewDirichlet = [];
1214   for j=1:size(N4D,1)
1215      if NOE(N4D(j,1),N4D(j,2))
1216          NewDirichlet = [NewDirichlet; N4D(j,1) NOE(N4D(j,1),N4D(j,2)); ...
1217                          NOE(N4D(j,1),N4D(j,2)) N4D(j,2)];
1218          NOE(N4D(j,1),N4D(j,2)) = 0;
1219      else
1220          NewDirichlet =[NewDirichlet; N4D(j,:)];
1221      end
1222   end
1223   N4D = NewDirichlet;
1224
1225   %Update Neumann edges
1226   NewNeumann = [];
1227   for j=1:size(N4N,1)
1228      if NOE(N4N(j,1),N4N(j,2))
1229          NewNeumann = [NewNeumann; N4N(j,1) NOE(N4N(j,1),N4N(j,2)); ...
1230                          NOE(N4N(j,1),N4N(j,2)) N4N(j,2)];
1231          NOE(N4N(j,1),N4N(j,2)) = 0;
1232      else
1233          NewNeumann =[NewNeumann; N4N(j,:)];
1234      end
1235   end
1236   N4N = NewNeumann;
1237
1238   %Green completion of CommonGrid
1239   [N4E,NOE] = greenCompletion(N4E,NOE,CommonGrid.Unused.N4E);
1240
1241   %Back to original data structures
1242   CommonGrid.C4N = C4N;
1243   CommonGrid.N4E = N4E;
1244   CommonGrid.N4D = N4D;
1245   CommonGrid.N4N = N4N;
1246   CommonGrid.F4E = F4E;
1247   CommonGrid.NOE = NOE;
```

```
1248
1249
1250
1251 %------------------------------------------------------------------------
1252 function [N4E,NOE] = greenCompletion(N4E,NOE,UnusedElements)
1253 %greenCompletion: green-refines all elements with a hanging node,
1254 %Remark: In contrast to other green completions this version may green
1255 %refine one element twice
1256
1257  Counter = size(N4E,1);
1258  HangingNodes = abs(NOE-NOE');
1259  [I1,I2] = find(HangingNodes);
1260  HangingNodesIdentifier = sparse(I1,I2,1,size(NOE,1),size(NOE,2));
1261
1262   for i=1:2
1263
1264       nze = setdiff(1:Counter,UnusedElements);
1265       Nnze = length(nze);
1266       a = N4E(nze,1:3)';
1267       I1 = reshape(a,3*Nnze,1);
1268       a = N4E(nze,[2 3 1])';
1269       I2 = reshape(a,3*Nnze,1);
1270       a = repmat(1:Nnze,3,1);
1271       Val = reshape(a,3*Nnze,1);
1272       ElementsOnEdges = sparse(I1,I2,Val,size(NOE,1),size(NOE,2));
1273       [I1,I2,ElementsToConsider] = ...
1274                               find(HangingNodesIdentifier.*ElementsOnEdges);
1275       ElementsToConsider = sort(nze(ElementsToConsider));
1276       %prelocation
1277       N4E = [N4E; zeros(nnz(HangingNodes)/2,5)];
1278
1279
1280       for j=1:nnz(ElementsToConsider)
1281           Element = ElementsToConsider(j);
1282
1283           for k=1:3
1284                if HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k))
1285                   HangingNode = ...
1286                       HangingNodes(N4E(Element,mod(k,3)+1),N4E(Element,k));
1287                   Counter = Counter + 1;
1288                   LocElement = N4E(Element,:);
1289                   N4E(Element,:) = ...
1290                      [HangingNode LocElement(rem(k+1,3)+1) LocElement(k)...
1291                                                          LocElement(4:5)];
1292                   N4E(Counter,:) = [HangingNode LocElement(rem(k,3)+1)...
1293                                LocElement(rem(k+1,3)+1) LocElement(4:5)];
1294
1295                   break;
1296               end
```

```
1297
1298            end
1299        end
1300    end
1301
1302    N4E = N4E(1:Counter,:);
1303
1304    %end of greenCompletion
1305 %-------------------------------------------------------------------------
1306 function I1 = ismemberP(A,B)
1307 %fast ismember('rows') for unique (3x2) matrices; I1 gives indices of A
1308 %where rows of B can be found
1309
1310     a= A-B;
1311     a = abs(a(:,1))+ abs(a(:,2));
1312     I1=find(~a);
1313     a= A-B([3 1 2],:);
1314     a = abs(a(:,1))+ abs(a(:,2));
1315     I2=find(~a);
1316     a= A-B([2 3 1],:);
1317     a = abs(a(:,1))+ abs(a(:,2));
1318     I3=find(~a);
1319     I1=[I1; I2; I3];
1320
1321     %end of ismemberP
1322 %-------------------------------------------------------------------------
1323 function I = ismemberP2(x,A)
1324 %fast ismember('rows') for x=(1x2) A=unique (3x2) matrix; I is
1325 %index in A which equals x or empty set
1326
1327     a= ones(size(A,1),1)*x-A;
1328     a = abs(a(:,1))+ abs(a(:,2));
1329     I = find(~a);
1330
1331     %end of ismemberP2
1332 %-------------------------------------------------------------------------
1333 function X = sortrowsP(X)
1334 %fast sort algorithm for 3x2 matrices
1335
1336 P = sortrowsP2(X(1:2,:));
1337
1338 [P2,Changed] = sortrowsP2(X([3 P(1)],:));
1339 if Changed
1340     [P2,Changed] = sortrowsP2(X([P(2) 3],:));
1341     if Changed
1342         P = [P(1) 3 P(2)];
1343     else
1344         P = [P(1) P(2) 3];
1345     end
```

```
1346 else
1347     P = [3; P];
1348 end
1349
1350 X = X(P,:);
1351
1352 %end of sortrowsP
1353 %-------------------------------------------------------------------------
1354 function [P,Changed] = sortrowsP2(X)
1355 %fast sort algorithm for 3x2 matrices subfunction of sortrowsP
1356
1357 if X(1,1)>X(2,1)
1358     P = [1;2];
1359     Changed = false;
1360 elseif X(1,1)<X(2,1)
1361     P = [2; 1];
1362     Changed = true;
1363 else
1364     if X(1,2)>X(2,2)
1365         P = [1; 2];
1366         Changed = false;
1367     else
1368         P = [2; 1];
1369         Changed = true;
1370     end
1371 end
1372
1373 %end of sortrowsP2
```

A.7 The Function *solveP*

```
0001 function [U,Solution] = solveP(Driver,Tolerance,Options)                %ok
0002 %   solveP: solves FEM analogue to parabolic equation
0003 %
0004 %       d/dt(u)-div(D(x,t)u) +C*Nabla(u)+R*u= f(x,t)  u=u(x,t) scalar fct
0005 %                                    u(x,t) = u_d(t) on Dirichlet-edge
0006 %                       D(x,t)*Nabla(u)*n(x) = g(x,t) on Neumann-edge
0007 %
0008 %   USAGE:   1)solveP(Driver) solves with data from Driver, tolerance from
0009 %               in Driver defined options file
0010 %            2)solveP(Driver,Tolerance) solves with data from Driver,
0011 %               Tolerance indicated local tolerance for spatial and
0012 %               temporal error estimator
0013 %            3)solveP(Driver,Tolerance,Options) options are not loaded as
0014 %               usual but taken from the input
0015 %            4)[U,Solution]=solveP(Driver,vargin) gives more data of the
0016 %             solution at final time, such as the grid for U(T) for example
0017 %
0018 %   INPUT ARGUMENTS
```

```
0019 %
0020 %   [Driver: String indicating the file where problem data, final time,etc
0021 %      is located]
0022 %
0023 %   Optional
0024 %   [Tolerance: local tolerance for time steps, bounds temporal and spatial
0025 %       error estimator]
0026 %
0027 %   [Options: option structure, see options.m]
0028 %
0029 %   OUTPUT ARGUMENTS:
0030 %
0031 %   [U: Discrete solution at final time T]
0032 %
0033 %   [Solution: Structure, stores all data for the final time step, i.e.
0034 %       U,Grid,k(timesteps),Eta(estimator),Error(optional),etc.]
0035 %
0036 %   See also refineGrid,coarsenGrid,commonGrid,parabolicSolver,options
0037 %
0038 %   Copyright (c) 2007 Philipp Wissgott
0039 %                      Vienna University of Technology
0040
0041 %************************ INITIALIZATION **********************************
0042
0043   set(0,'ShowHiddenHandles','on')
0044   delete(get(0,'Children'))
0045   format long;
0046   CPUTimeAll = cputime;
0047   T = feval(Driver,'TimeInterval');
0048
0049 %*************************** OPTIONS **************************************
0050
0051   OptionFile = feval(Driver,'OptionFile');
0052   Options.ScreenSize = get(0,'ScreenSize'); %visualisation
0053   if nargin==1
0054      Options = feval(OptionFile,'LoadOptions','General',Driver,T(1),T(2));
0055      Options.OptionFile = OptionFile;
0056      epsilon = Options.Tolerance;
0057   elseif nargin==2
0058      Options = feval(OptionFile,'LoadOptions','General',Driver,T(1),T(2));
0059      Options.OptionFile = OptionFile;
0060      epsilon = Tolerance;
0061   else
0062      epsilon = Tolerance;
0063   end
0064
0065 %************************** MAIN ******************************************
0066
0067   %Initialize logging
```

```
0068  if Options.LogAll
0069      LogFile=fopen(Options.LogFile,'wt');
0070      Text1 = sprintf('tol=%g,time interval=[%g,%g],Theta=%g, ',...
0071                                          epsilon,T(1),T(2),Options.Theta);
0072      Text2 = ...
0073         sprintf('dependencies:D=%1.0f C=%1.0f R=%1.0f F=%1.0f G=%1.0f',...
0074                                                  Options.Dependence.D,...
0075                                Options.Dependence.C,Options.Dependence.R,...
0076                                  Options.Dependence.F,Options.Dependence.G);
0077      Text = strcat('Start computation: ',Text1,Text2);
0078      fprintf(LogFile,Text);
0079      fclose(LogFile);
0080      LogFile=fopen(Options.LogFile,'at');
0081  end
0082
0083  %Load initial grid
0084  Grid(1) = feval(Driver,'InitialGrid');
0085  NOFN = size(Grid(1).C4N,1)-size(Grid(1).Unused.C4N,1)...
0086                                                  -nnz(unique(Grid(1).N4D));
0087  if Options.LogAll
0088      fprintf(LogFile,'\nLoad initial triangulation...');
0089  end
0090
0091  %Get initial error indicator
0092  [Options.InitialError,Grid(1).M,Grid(1).DoubleArea] = ...
0093    feval(OptionFile,'InitialError','InitialError',Grid(1),Driver,Options);
0094  while Options.InitialError>epsilon
0095     Grid(1) = refineGrid(Grid(1),'All',Driver);
0096     [Options.InitialError,Grid(1).M,Grid(1).DoubleArea] = ...
0097    feval(OptionFile,'InitialError','InitialError',Grid(1),Driver,Options);
0098     NOFN = size(Grid(1).C4N,1)-size(Grid(1).Unused.C4N,1)...
0099                                                  -nnz(unique(Grid(1).N4D));
0100     if NOFN>Options.MaximumNumberOfFreeNodes
0101         error('Too many free nodes. Decrease tolerance!')
0102     end
0103  end
0104
0105  %try to satisfy characteristic length
0106  if Options.CharacteristicLength
0107       Vertices = Grid(1).C4N(Grid(1).N4E(1,1:3),:);
0108       Diameter = max([norm(Vertices(2,:)-Vertices(1,:)) ...
0109                       norm(Vertices(3,:)-Vertices(2,:)) ...
0110                       norm(Vertices(1,:)-Vertices(3,:))]);
0111       while Diameter>Options.CharacteristicLength
0112
0113           Grid(1) = refineGrid(Grid(1),'All',Driver);
0114           Vertices = Grid(1).C4N(Grid(1).N4E(1,1:3),:);
0115           Diameter = max([norm(Vertices(2,:)-Vertices(1,:)) ...
0116                       norm(Vertices(3,:)-Vertices(2,:)) ...
```

```
0117                        norm(Vertices(1,:)-Vertices(3,:))]);
0118            NOFN = size(Grid(1).C4N,1)-size(Grid(1).Unused.C4N,1)...
0119                                                  -nnz(unique(Grid(1).N4D));
0120            fprintf(LogFile,'\nTry to satisfy characteristic',...
0121                                  'length: current NOFN=%g, diameter=%g',...
0122                                                       NOFN,Diameter);
0123            if NOFN>Options.MaximumNumberOfFreeNodes
0124               error('Too many free nodes. Decrease tolerance!')
0125            end
0126         end
0127   end
0128
0129   %Set initial time step
0130   try
0131       InitialTau = feval(Driver,'InitialTimeStep');
0132   catch
0133       InitialTau = epsilon;
0134   end
0135
0136   %solve
0137   Solution = solving(Grid(1),T,epsilon,InitialTau,Driver,Options);
0138
0139   %Option:CompareToGivenSolution
0140   if Options.Compare
0141       try
0142          Solution.Error = ...
0143                    realError(Solution.Grid,Solution.U,T(2),Driver,Options);
0144       catch
0145          Solution.Error = 0;
0146          warning('No or erroneous analytic solution. Check option file!')
0147       end
0148   end
0149
0150   %Set output
0151   U = Solution.U;
0152
0153   %Finalize logging
0154   if Options.LogAll
0155       LogFile=fopen(Options.LogFile,'at');
0156       Text1 = sprintf('\nComputation complete: Eta=%g,',Solution.Eta);
0157       Text2 = sprintf('elapsed time=%gs',cputime-CPUTimeAll);
0158       Text = strcat(Text1,Text2);
0159       fprintf(LogFile,Text);
0160   end
0161
0162   %visualization
0163   if Options.DisplayOutput
0164
0165      Text1 = sprintf('sol. for %s on time interval=[%g,%g], ',...
```

```
0166                                                          Driver,T(1),T(2));
0167      Text2 = sprintf('theta=%g,Tol=%g, Eta=%g',...
0168                                        Options.Theta,epsilon,Solution.Eta);
0169      Title = strcat(Text1,Text2);
0170      show(Solution.Grid,U,Title,1);
0171      if Options.SaveOutput
0172         print Solution1 -depsc;
0173      end
0174      newFigure(2)
0175      plot(Solution.k,Solution.NOFN,'-x')
0176      Text1 = sprintf('t(i) versus NOFN for %s on [%.1f,%.1f]',...
0177                                                          Driver,T(1),T(2));
0178      Text2 = sprintf('for theta=%.2f',Options.Theta);
0179      title(strcat(Text1,Text2))
0180      xlabel('t(i)')
0181      ylabel('NOFN')
0182      if Options.SaveOutput
0183         print Solution2 -depsc;
0184      end
0185      if Options.Compare
0186          Title=sprintf('Analytic sol. for %s on time interval=[%g,%g],',...
0187                                                          'Error=%g',...
0188                                          Driver,T(1),T(2),Solution.Error);
0189          U = feval(Driver,'u(x,t)',Solution.Grid.C4N,T(2));
0190          show(Solution.Grid,U,Title,3);
0191      end
0192      if Options.LogAll
0193         newFigure(4);
0194         displayLog;
0195      end
0196   end
0197
0198   if Options.SaveOutput
0199       save Solution Solution
0200   end
0201
0202    if Options.LogAll
0203        fclose(LogFile);
0204    end
0205
0206 % end of solveP
0207 %-------------------------------------------------------------
0208 function Solution = ...
0209                     solving(InitialGrid,T,epsilon,InitialTau,Driver,Options)
0210 %actual solving
0211
0212   %initial timestep
0213   k = T(1);
0214
```

```
0215  %initial triangulation
0216  Grid(1) = InitialGrid;
0217  Grid(2) = InitialGrid;
0218
0219  %initial number of free nodes
0220  NOFN = size(Grid(1).C4N,1)-size(Grid(1).Unused.C4N,1)...
0221                                                  -nnz(unique(Grid(1).N4D));
0222  NOFNAll = NOFN;
0223
0224  %get initial error estimator
0225  [Options.InitialError,Grid(1).M,Grid(1).DoubleArea] = ...
0226                                 feval(Options.OptionFile,'InitialError',...
0227                                     'InitialError',Grid(1),Driver,Options);
0228  Eta = Options.InitialError^2;
0229
0230  %Logging
0231  if Options.LogAll
0232      LogFile=fopen(Options.LogFile,'at');
0233      Text1 = sprintf('\nStart solving: initial error estimator=%g,',Eta);
0234      Text2 = sprintf(' initial NOFN=%g',NOFN);
0235      fprintf(LogFile,strcat(Text1,Text2));
0236  end
0237
0238  %set initial configuration
0239  t = [T(1) min(InitialTau,(T(2)-T(1))/2)];
0240  i=2;
0241  Solution.Singularity = 0;
0242  %initial u=\Pi_0 u_0 (L^2-projection)
0243  Uold = feval(Driver,'u(x,t)',Grid(1).C4N,T(1));
0244
0245  %optional: obtain data for slices of the domain
0246  if ~isempty(Options.Cut)
0247      Value = Options.Cut(1,:)*Options.Cut(2,:)';
0248      Nodes = find(Grid(1).C4N*Options.Cut(2,:)'==Value);
0249      CutData(1).Coordinates = Grid(1).C4N(Nodes,:);
0250      CutData(1).Values = Uold(Nodes);
0251      CutData(1).Time = t(1);
0252      CutData(1).Axis = Options.Cut;
0253      save CutData CutData
0254  end
0255
0256  %start solving
0257  while t(i)<=T(2)
0258
0259      i = length(t);
0260
0261      %optional: debug modus
0262      if Options.SaveOutput == 2
0263           save debug1
```

```
0264            testConsistency(Grid(2),feval(Driver,'Area'))
0265        end
0266
0267        %solve first time to obtain time step information
0268        [U,Grid(2)] = ...
0269                    parabolicSolver(Grid(1),Grid(2),Uold,t(i-1),t(i),Driver);
0270        Eta_i.Temporal = ...
0271                      etaTemporal(t(i-1),t(i),Driver,Grid(1),Grid(2),Uold,U);
0272
0273        Oldtau = t(i)-t(i-1);
0274        %Set new time step
0275        t(i) = min(T(2),t(i-1)+ (t(i)-t(i-1))* ...
0276                          (epsilon/(Eta_i.Temporal.TemporalErrorEstimator)));
0277
0278        %Logging
0279        if Options.LogAll
0280           fprintf(LogFile,...
0281                       '\nAt %g: Changed time step size from %g to %g...',...
0282                                                 t(i-1),Oldtau,(t(i)-t(i-1)));
0283        end
0284
0285        %solve second time to obtain spatial error estimator information
0286        [U,Grid(2)] = ...
0287                  parabolicSolver(Grid(1),Grid(2),Uold,t(i-1),t(i),Driver,1);
0288        [MarkedElements,Eta_i.Spatial,ElementIndicators] = ...
0289                       etaSpatial(Grid(1),Grid(2),Uold,U,t(i-1),t(i),Driver);
0290
0291        %try to satisfy spatial estimator condition
0292        Refined = 0;
0293        while Eta_i.Spatial.SpatialErrorEstimator>epsilon
0294
0295            OldNOFN = size(Grid(2).C4N,1)-size(Grid(2).Unused.C4N,1)...
0296                                                  -nnz(unique(Grid(2).N4D));
0297            %refine marked elements
0298            [Grid(2),LogStruct] = refineGrid(Grid(2),MarkedElements,Driver);
0299            Refined = 1;
0300            NOFN = size(Grid(2).C4N,1)-size(Grid(2).Unused.C4N,1)...
0301                                                  -nnz(unique(Grid(2).N4D));
0302            %Logging
0303            if Options.LogAll
0304                fprintf(LogFile,...
0305                      '\nAt %g: Refined grid from %g to %g free nodes...',...
0306                                                       t(i-1),OldNOFN,NOFN);
0307            end
0308
0309            %optional debug modus
0310            if Options.SaveOutput == 2
0311                save debug1
0312                testConsistency(Grid(2),feval(Driver,'Area'))
```

```
0313            end
0314
0315            %solve to continue or to obtain new marked element
0316            [U,Grid(2)] = ...
0317                    parabolicSolver(Grid(1),Grid(2),Uold,t(i-1),t(i),Driver);
0318            [MarkedElements,Eta_i.Spatial,ElementIndicators] = ...
0319                      etaSpatial(Grid(1),Grid(2),Uold,U,t(i-1),t(i),Driver);
0320
0321        end
0322
0323        %Accept this time step and solution
0324        k = [k t(i)];
0325        NOFNAll = [NOFNAll NOFN];
0326        %compute temporal error estimator
0327        Eta_i.Temporal = ...
0328                       etaTemporal(t(i-1),t(i),Driver,Grid(1),Grid(2),Uold,U);
0329        %Update global error estimator
0330        Eta = Eta + (t(i)-t(i-1))...
0331                             *(Eta_i.Spatial.SpatialErrorEstimator^2 +...
0332                               Eta_i.Temporal.TemporalErrorEstimator^2);
0333        %Logging
0334        if Options.LogAll
0335            Text1 = '\nLocal computation complete: ';
0336            Text2 = sprintf('local error estimator=%g...',...
0337                         (Eta_i.Spatial.SpatialErrorEstimator+...
0338                             Eta_i.Temporal.TemporalErrorEstimator^2)^(1/2));
0339            fprintf(LogFile,strcat(Text1,Text2));
0340        end
0341
0342        %stop if algorithm reached end of time interval
0343        if t(i)==T(2)
0344
0345            %save output
0346            Solution.k = k;
0347            Solution.NOFN = NOFNAll;
0348            Solution.Grid = Grid(2);
0349            Solution.Eta = sqrt(Eta);
0350            Solution.U = U;
0351            Solution.Tolerance = epsilon;
0352            Solution.T = T;
0353            Solution.Options = Options;
0354            Solution.Driver = Driver;
0355
0356            %optional: save slice data
0357            if ~isempty(Options.Cut)
0358                try
0359                   load CutData
0360                catch
0361                end
```

```
0362                Value = Options.Cut(1,:)*Options.Cut(2,:)';
0363                Nodes = find(Grid(2).C4N*Options.Cut(2,:)'==Value);
0364                CutData(i).Coordinates = Grid(2).C4N(Nodes,:);
0365                CutData(i).Values = U(Nodes);
0366                CutData(i).Time = t(i);
0367                CutData(i).Axis = Options.Cut;
0368                save CutData CutData
0369            end
0370            %optional: save output
0371            if Options.SaveOutput
0372               save Solution Solution;
0373            end
0374            if Options.LogAll
0375                fclose(LogFile);
0376            end
0377            return;
0378        end
0379
0380        %Update Grid and U
0381        Grid(1) = Grid(2);
0382        Uold = U;
0383
0384        %coarsening, only if time step size increased
0385        if i>2 && (t(i)-t(i-1))>(t(i-1)-t(i-2))
0386            OldNOFN = size(Grid(2).C4N,1)-size(Grid(2).Unused.C4N,1)...
0387                                                    -nnz(unique(Grid(2).N4D));
0388            Grid(2) = coarsenGrid(Grid(2),MarkedElements,Driver);
0389            NOFN = size(Grid(2).C4N,1)-size(Grid(2).Unused.C4N,1)...
0390                                                   -nnz(unique(Grid(2).N4D));
0391            %logging
0392            if Options.LogAll && OldNOFN>NOFN
0393                Text1 = sprintf('\nAt %g: Coarsenend grid from ',t(i-1));
0394                Text2 = sprintf('%g to %g free nodes...',OldNOFN,NOFN);
0395                fprintf(LogFile,strcat(Text1,Text2));
0396            end
0397        else
0398            Grid(2) = Grid(2);
0399        end
0400
0401        %first guess time step size for the next time step
0402        t(i+1) = min(T(2),t(i) + (t(i)-t(i-1)));
0403
0404        %logging
0405        if Options.LogAll
0406            fprintf(LogFile,'\nNext time step=%g...',t(i));
0407        end
0408
0409        %store output
0410        Solution.k = k;
```

```
0411        Solution.NOFN = NOFNAll;
0412        Solution.Grid = Grid(2);
0413        Solution.Eta = sqrt(Eta);
0414        Solution.U = U;
0415        Solution.Tolerance = epsilon;
0416        Solution.T = T;
0417        Solution.Options = Options;
0418        Solution.Driver = Driver;
0419
0420        %optional: save slice data
0421        if ~isempty(Options.Cut)
0422            try
0423               load CutData
0424            catch
0425            end
0426            Value = Options.Cut(1,:)*Options.Cut(2,:)';
0427            Nodes = find(Grid(2).C4N*Options.Cut(2,:)'==Value);
0428            CutData(i).Coordinates = Grid(2).C4N(Nodes,:);
0429            CutData(i).Values = U(Nodes);
0430            CutData(i).Time = t(i);
0431            CutData(i).Axis = Options.Cut;
0432            save CutData CutData
0433        end
0434
0435        %optional: save output
0436        if Options.SaveOutput
0437           save Solution Solution;
0438        end
0439
0440        %next time step
0441
0442    end
0443
0444 %   end of solving
0445 %-------------------------------------------------------------------------
0446 function RealError = realError(Grid,Uh,T,Driver,Options)
0447 %real error in energy norm \!e\! at time T
0448
0449     Sum1 = 0;
0450     Sum2 = 0;
0451     nze = find(Grid.N4E(:,1));
0452     NablaEta = zeros(3,2);
0453
0454     for j=1:size(nze,1)
0455
0456        Nodes = Grid.N4E(nze(j),1:3);
0457        Vertices = Grid.C4N(Nodes,:);
0458        Uhloc = Options.QBasisfct*Uh(Nodes);
0459        DoubleArea = det([ones(1,3);Vertices']);
```

```
0460        NablaEta(:,1) = Vertices([2 3 1],2) - Vertices([3 1 2],2);
0461        NablaEta(:,2) = Vertices([3 1 2],1) - Vertices([2 3 1],1);
0462        NablaEta = NablaEta/DoubleArea;
0463
0464          NablaUh = Uh(Nodes)'*NablaEta;
0465        U = feval(Driver,'u(x,t)',...
0466                              ones(size(Options.QWeights,1),1)*Vertices(1,:)...
0467                             + Options.QNodes*[Vertices(2,:)-Vertices(1,:);
0468                                               Vertices(3,:)-Vertices(1,:)],T);
0469        NablaU = feval(Driver,'Nablau(x,t)',...
0470                              ones(size(Options.QWeights,1),1)*Vertices(1,:)...
0471                              + Options.QNodes*[Vertices(2,:)-Vertices(1,:);
0472                                               Vertices(3,:)-Vertices(1,:)],T);
0473        %Norm |||U-Uh|||_K^2
0474        a = (NablaU(:,1)-NablaUh(:,1)).^2 + (NablaU(:,2)-NablaUh(:,2)).^2;
0475        Sum1 =  Sum1 + det([ones(1,3);Vertices'])*Options.QWeights'*a;
0476        Sum2 =  Sum2 + det([ones(1,3);Vertices'])*Options.QWeights'...
0477                                                              *(U-Uhloc).^2;
0478
0479    end
0480
0481    RealError = sqrt(Options.Lambda*Sum1 + Options.Beta*Sum2);
0482
0483 %  end of realError
```

A.8 The Function *options*

```
0001 function [out1,out2,out3] = options(flag1,flag2,varargin)
0002 % options: set options for solveP
0003
0004    switch(flag1)
0005
0006        %General options
0007        case 'General'
0008             switch(flag2)
0009                 case 'Tolerance'
0010                     %local tolerance; double >0
0011                     out1 = 0.5;
0012                  case 'MaximumNumberOfFreeNodes'
0013                     %integer
0014                      out1 = 3e6;
0015                 case 'ShowTime';
0016                     %show computation time for all major
0017                     %subfunctions=1, 0 otherwise
0018                     out1 = 0;
0019                 case 'LogAll'
0020                     %Global logging=1, 0 otherwise
0021                     out1 = 1;
0022                 case 'LogFile'
```

```
0023                    %file name for global logging, string
0024                    out1 = 'Log.txt';
0025                case 'CompareToGivenSolution'
0026                    %Compute error in energy norm of the
0027                    %dicrete solution at final time the values stored at
0028                    %'u(x,t)' in the driver=1, 0 otherwise
0029                    out1 = 1;
0030                case 'SaveOutput'
0031                    %Save ouput=1, debug level=2, 0 otherwise
0032                    out1 = 1;
0033                case 'DisplayOutput'
0034                    %display discrete solution at final time=1, 0 otherwise
0035                    out1 = 0;
0036                case 'Cut'
0037                    %Set slice parameter: out1(1,:)=point, out1(2,:)=normal
0038                    %vector of slicing line, [] for no slicing
0039                    out1 = [];%example:[50 0;1 0]
0040                otherwise
0041                   error('+++ Unknown flag');
0042             end
0043
0044        %FEM-solver Options
0045        case 'Solver'
0046            switch(flag2)
0047                case 'Theta'%time discretization parameter double in [1/2,1]
0048                    out1 = 1/2;
0049                case 'QuadraturNodes'
0050                    %double nx2 array witn n the number of
0051                    %nodes
0052                    %out1 = [0.5 0;0.5 0.5;0 0.5]; %3 points middle edges
0053                    %out1 = [0 0;1 0;0 1];  %vertices rule
0054                    out1 = [1 1]/3; %center of gravity rule
0055                case 'QuadraturWeights'
0056                    %vertical nx1 vector witn n the number of weights
0057                    %out1 = [1 1 1]'/6; %3 points middle edges
0058                    %out1 = [1 1 1]'/6; %vertices rule
0059                    out1 = [1/2]'; %center of gravity rule
0060                case 'DisplayOutput'
0061                    %diplay discrete solution for each time step=1, 0 otherw.
0062                    out1 = 1;
0063                case 'ShowTime'
0064                    %show computation time for parabolicSolver=1, 0 otherwise
0065                    out1 = 0;
0066                otherwise
0067                    error('+++ Unknown flag');
0068            end
0069
0070        %Indicator options(temporal & spatial)
0071        case 'Indicator'
```

```
0072            switch(flag2)
0073                case 'DisplayOutput'
0074                    %display output of etaSpatial=1, 0 otherwise; the output
0075                    %is a colored figure which shows the element indicators
0076                    out1 = 0;
0077                case 'ShowTime'
0078                    %display computation time=1,0 otherwise
0079                    out1 = 0;
0080                case  'Theta'
0081                    %indicated how many elements are refined/corsened,
0082                    %double [0,1], 0 means uniform refinement/ no coarsening,
0083                    %1 means only the element with the largest indicator
0084                    %value is refined
0085                    out1 = 0.5;
0086                otherwise
0087                    error('+++ Unknown flag');
0088            end
0089
0090        %Refinement options
0091        case 'Refinement'
0092            switch(flag2)
0093                case 'SaveOutput'
0094                    %save the all data from refineGrid in
0095                    %refinementOutput.mat=1, 0 otherwise
0096                    out1 = 0;
0097                case 'DisplayOutput'
0098                    %diplays 2D grid output of the resulting triangulation=1,
0099                    %0 otherwise
0100                    out1 = 0;
0101                case 'ShowTime'
0102                    %show computation time for refineGrid=1, 0 otherwise
0103                    out1 = 0;
0104                otherwise
0105                    error('+++ Unknown flag');
0106            end
0107
0108        %Coarsening options
0109        case 'Coarsening'
0110            switch(flag2)
0111                case 'DisplayOutput'
0112                    %diplays 2D grid output of the resulting triangulation=1,
0113                    %0 otherwise
0114                    out1 = 0;
0115                case 'ShowTime'
0116                    %show computation time for coarsenGrid=1, 0 otherwise
0117                    out1 = 0;
0118                case 'CoarseningLevel'
0119                    %Set coarsening level to 'Uniform' for uniform
0120                    %coarsening, for all other inputs adaptive coarsening is
```

```
0121                    %conducted
0122                    out1 = 'Adaptive';
0123                otherwise
0124                    error('+++ Unknown flag');
0125            end
0126
0127        %Common grid Options
0128        case 'CommonGrid'
0129            switch(flag2)
0130                case 'DisplayOutput'
0131                    %diplays 2D grid output of the resulting triangulation=1,
0132                    %0 otherwise
0133                    out1 = 0;
0134                case 'ShowTime'
0135                    %show computation time for commonGrid=1, 0 otherwise
0136                    out1 = 0;
0137                otherwise
0138                    error('+++ Unknown flag');
0139            end
0140
0141        case 'LoadOptions'
0142            %load options for flag2=driver
0143            OptionFile = feval(varargin{1},'OptionFile');
0144
0145            %load general options
0146            try
0147                out1.Tolerance = feval(OptionFile,'General','Tolerance');
0148            catch
0149                out1.Tolerance = 1;
0150            end
0151            try
0152                out1.LogAll = feval(OptionFile,'General','LogAll');
0153            catch
0154                out1.LogAll = 0;
0155            end
0156            if out1.LogAll
0157                out1.LogFile = feval(OptionFile,'General','LogFile');
0158            end
0159            try
0160                out1.MaximumNumberOfFreeNodes = ...
0161                      feval(OptionFile,'General','MaximumNumberOfFreeNodes');
0162            catch
0163                out1.MaximumNumberOfFreeNodes = 3e6;
0164            end
0165            try
0166                out1.Cut = feval(OptionFile,'General','Cut');
0167            catch
0168                out1.Cut=[];
0169            end
```

```
0170            try
0171               out1.CharacteristicLength = ...
0172                                   feval(varargin{1},'CharacteristicLength');
0173            catch
0174                out1.CharacteristicLength = 0;
0175            end
0176            try
0177                out1.SaveOutput = feval(OptionFile,flag2,'SaveOutput');
0178            catch
0179                out1.SaveOutput = 0;
0180            end
0181            try
0182                out1.DisplayOutput = feval(OptionFile,flag2,'DisplayOutput');
0183            catch
0184                out1.DisplayOutput = 0;
0185            end
0186
0187            if feval(OptionFile,'General','ShowTime')
0188                out1.ShowTime = 1;
0189            else
0190                out1.ShowTime = feval(OptionFile,flag2,'ShowTime');
0191            end
0192            out1.Compare = feval(OptionFile,'General','CompareToGivenSolution');
0193            out1.Theta = feval(OptionFile,'Solver','Theta');
0194            out1.Lambda = feval(varargin{1},'Lambda');
0195            out1.Beta = feval(varargin{1},'Beta');
0196            out1.IndicatorTheta = feval(OptionFile,'Indicator','Theta');
0197
0198            %load quadratur data
0199            [out1.QNodes,out1.QWeights,out1.QBasisfct] = quadratur(varargin{1});
0200
0201            %Test dependencies
0202            [out1.Dependence.D,out1.Value.D] = ...
0203                testDependence('D(x,t)',varargin{2},varargin{3},varargin{1});
0204            [out1.Dependence.C,out1.Value.C] = ...
0205                testDependence('c(x,t)',varargin{2},varargin{3},varargin{1});
0206            [out1.Dependence.R,out1.Value.R] = ...
0207                testDependence('r(x,t)',varargin{2},varargin{3},varargin{1});
0208            [out1.Dependence.F,out1.Value.F] = ...
0209                testDependence('f(x,t)',varargin{2},varargin{3},varargin{1});
0210            [out1.Dependence.G,out1.Value.G] = ...
0211                testDependence('g(x,t)',varargin{2},varargin{3},varargin{1});
0212
0213        case 'InitialError'
0214
0215            nze = find(varargin{1}.N4E(:,1));
0216            nzc = ...
0217                setdiff((1:size(varargin{1}.C4N,1))',varargin{1}.Unused.C4N);
0218
```

```
0219          %Assembly of mass matrix for varargin{1}=Grid
0220          a = [varargin{1}.N4E(nze,1) varargin{1}.N4E(nze,1) ...
0221               varargin{1}.N4E(nze,1) varargin{1}.N4E(nze,2) ...
0222               varargin{1}.N4E(nze,2) varargin{1}.N4E(nze,2) ...
0223               varargin{1}.N4E(nze,3) varargin{1}.N4E(nze,3) ...
0224                                      varargin{1}.N4E(nze,3)]';
0225          I1 = reshape(a,9*nnz(nze),1);
0226          a = [varargin{1}.N4E(nze,1:3) varargin{1}.N4E(nze,1:3)...
0227                                              varargin{1}.N4E(nze,1:3)]';
0228          I2 = reshape(a,9*nnz(nze),1);
0229          I3 = reshape(varargin{1}.N4E(nze,1:3)',3*nnz(nze),1);
0230          %Assembly of mass matrix for Grid
0231          A = varargin{1}.C4N(varargin{1}.N4E(nze,1),:);
0232          B = varargin{1}.C4N(varargin{1}.N4E(nze,2),:);
0233          C = varargin{1}.C4N(varargin{1}.N4E(nze,3),:);
0234          %Assembly of areas
0235          DoubleArea = B(:,1).*C(:,2) + C(:,1).*A(:,2) + A(:,1).*B(:,2) ...
0236                  -  B(:,1).*A(:,2) - C(:,1).*B(:,2) - A(:,1).*C(:,2) ;
0237          %Assembly of mass matrix
0238          RefM = [2 1 1 1 2 1 1 1 2]/24;
0239          LocMassMatrices = reshape(RefM'*DoubleArea',9*size(A,1),1);
0240          M = sparse(I1,I2,LocMassMatrices,...
0241                         size(varargin{1}.C4N,1),size(varargin{1}.C4N,1));
0242          clear I1 I2 LocMassMatrices
0243
0244          %L2-projection of u^{(0)}
0245          Projection.U_0 = zeros(size(varargin{1}.C4N,1),1);
0246          Values = zeros(3,size(A,1));
0247          for i=1:size(varargin{3}.QWeights,1)
0248             Values = Values + ...
0249                    varargin{3}.QWeights(i)*varargin{3}.QBasisfct(i,:)'*...
0250            (feval(varargin{2},'u(x,t)',A+(B-A)*varargin{3}.QNodes(i,1)+...
0251                           (C-A)*varargin{3}.QNodes(i,2),0).*DoubleArea)';
0252          end
0253          Values = reshape(Values,3*size(A,1),1);
0254          RHS = ...
0255            sparse(I3,ones(size(I3,1),1),Values,size(varargin{1}.C4N,1),1);
0256          M2 = M(nzc,nzc);
0257          RHS2 = RHS(nzc);
0258
0259          if nnz(nzc)>1e5 %large dimension mode
0260             save temp;
0261             save temp2 M2 RHS2;
0262             clear all;
0263             load temp2;
0264
0265             X = M2\RHS2;
0266
0267             load temp
```

```
0268              Projection.U_0(nzc) = X;
0269           else
0270              Projection.U_0(nzc) = M2\RHS2;
0271           end
0272
0273           %Compute intial error
0274           out1 = 0;
0275           for j=1:size(nze,1)
0276               Nodes = varargin{1}.N4E(nze(j),1:3);
0277               Vertices = varargin{1}.C4N(Nodes,:);
0278               Uh_0 = varargin{3}.QBasisfct*Projection.U_0(Nodes);
0279               U_0 = feval(varargin{2},'u(x,t)',...
0280                       ones(size(varargin{3}.QWeights,1),1)*Vertices(1,:)...
0281                      + varargin{3}.QNodes*[Vertices(2,:)-Vertices(1,:);
0282                                            Vertices(3,:)-Vertices(1,:)],0);
0283               a = U_0 - Uh_0;
0284               out1 = out1 ...
0285                      + det([1 1 1 ; Vertices'])*varargin{3}.QWeights'*a.^2;
0286           end
0287
0288           out1 = sqrt(out1);
0289           out2 = M;
0290           out3 = DoubleArea;
0291
0292   otherwise
0293       error('+++ Unknown flag');
0294   end
0295
0296 %   end of options
0297 %--------------------------------------------------------------------------
0298 function [Dependence,Value] = testDependence(Input,Told,T,Driver)
0299 %testDependence: obtain dependencies and values for Input string
0300 % -1=trivial,0 constant,1 time dependent,2 spatially dependent,3 time and
0301 % spatially dependent
0302
0303     Dependence = 0;
0304     Value = 0;
0305     try
0306        Value = feval(Driver,Input,{},{});
0307        if ~nnz(Value)
0308            Dependence = -1;
0309        end
0310     catch
0311        try
0312           Value = feval(Driver,Input,{},Told);
0313           Value = [Value; feval(Driver,Input,{},T)];
0314        catch
0315            Dependence = Dependence + 2;
0316        end
```

```
0317        try
0318           ValueT = feval(Driver,Input,[0 0],{});
0319           if isempty(ValueT)
0320               Dependence = Dependence + 1;
0321           end
0322        catch
0323            Dependence = Dependence + 1;
0324        end
0325     end
0326
0327 %     end of testDependence
0328 %-------------------------------------------------------------------------
0329 function [Nodes,Weights,Basisfct] = quadratur(Driver)
0330 %load quadratur options predefined by user in optionsfile of the driver
0331
0332   OptionFile = feval(Driver,'OptionFile');
0333   Weights = feval(OptionFile,'Solver','QuadraturWeights');
0334   Nodes =  feval(OptionFile,'Solver','QuadraturNodes');
0335
0336   if size(Weights,1)==size(Nodes,1)
0337       Basisfct = zeros(3,max(size(Weights,1),1));
0338       Basisfct(:,1) = [1 1 1]'/3;
0339       if nnz(Weights)
0340           Basisfct = [ones(size(Weights,1),1)-Nodes(:,1)-Nodes(:,2) ...
0341                                                   Nodes(:,1) Nodes(:,2)];
0342       end
0343
0344   else
0345      Text1 = 'Number of quadratur weights has to match';
0346      Text2 = ' to number of nodes. Check options-file!';
0347      error(strcat(Text1,Text2));
0348   end
0349
0350      %end of quadratur
```

A.9 The Function *Model1Driver*

```
0001 function out1 = Model1Driver(flag,varargin)
0002 %Driver for Model1: d_t u-Laplace(u)+u=2tx^2 - 2*t^2 + t^2*x^2
0003 %includes all accordant problem data
0004
0005 if nargin>1
0006    %rename coordinates input
0007    X = varargin{1};
0008 end
0009
0010 if nargin==3
0011    %rename time input
0012    t = varargin{2};
```

```
0013 end
0014
0015 switch(flag)
0016     case 'InitialGrid'
0017         %Load initial grid from coordinates.dat,elements.dat,dirichlet.dat
0018         %and neumann.dat
0019
0020         %nx2 matrix, vertices
0021         try
0022             load coordinates.dat
0023             out1.C4N = coordinates;
0024         catch
0025             out1.C4N = [];
0026         end
0027
0028         %nx3/nx4 matrix, elements
0029         try
0030             load elements.dat;
0031             out1.N4E =[elements zeros(size(elements,1),2)];
0032         catch
0033             out1.N4E = [];
0034         end
0035         %nx2 matrix, dirichlet edge
0036         try
0037             load dirichlet.dat;
0038             out1.N4D = dirichlet;
0039         catch
0040             out1.N4D = [];
0041         end
0042
0043         %nx2 matrix, neumann edge
0044         try
0045             load neumann.dat;
0046             out1.N4N = neumann;
0047         catch
0048             out1.N4N = [];
0049         end
0050
0051         %initial grid values
0052         out1.Unused.N4E = [];
0053         out1.Unused.C4N = [];
0054         out1.Unused.F4E = [];
0055         out1.NOE = sparse(size(out1.C4N,1),size(out1.C4N,1));
0056         out1.F4E = [];
0057         out1.M = [];
0058         out1.H = [];
0059         out1.b = [];
0060         out1.DoubleArea = [];
0061
```

```
0062     case 'u(x,t)' %initial U, boundary U nx1
0063             out1 = t^2*X(:,1).^2;
0064
0065     case 'Nablau(x,t)' %Nabla u ;nx2 horizontal vector
0066         out1 =  2*t^2*[X(:,1) zeros(max(1,size(X,1)),1)];
0067
0068     case 'f(x,t)' %nx1 ,rhs
0069         out1 =2*t*X(:,1).^2 ...
0070               - 2*t^2*ones(max(1,size(X,1)),1) ...
0071               + t^2*X(:,1).^2;
0072
0073     case 'df(x,t)/dt' %nx1
0074         out1 =2*X(:,1).^2-4*t*...
0075                 ones(max(1,size(X,1)),1)...
0076                 + 2*t*X(:,1).^2;
0077
0078     case 'g(x,t)' %nx1 ,Neumann-edge function
0079         out1 =zeros(max(1,size(X,1)),1);
0080
0081     case 'dg(x,t)/dt' %nx1
0082         out1 =zeros(max(1,size(X,1)),1);
0083
0084     case 'D(x,t)' %2nx2-matrix
0085         out1 = [ones(max(1,size(X,1)),1) ...
0086                 zeros(max(1,size(X,1)),1);...
0087                 zeros(max(1,size(X,1)),1) ...
0088                 ones(max(1,size(X,1)),1)];
0089
0090     case 'derD(x,t)' %nx2 horizontal vector,
0091         %derivative vector:[dD11/dx+dD21/dy dD12/dx+dD22/dy]
0092         out1 = [zeros(max(1,size(X,1)),1) ...
0093                 zeros(max(1,size(X,1)),1)];
0094
0095      case 'dD(x,t)/dt' %2nx2-matrix
0096         out1 = [zeros(max(1,size(X,1)),1) ...
0097                 zeros(max(1,size(X,1)),1);...
0098                 zeros(max(1,size(X,1)),1) ...
0099                 zeros(max(1,size(X,1)),1)];
0100
0101     case 'c(x,t)' % nx2 horizontal vector
0102         out1 = [zeros(max(1,size(X,1)),1) ...
0103                 zeros(max(1,size(X,1)),1) ];
0104
0105      case 'dc(x,t)/dt' %nx2 horizontal vector
0106         out1 = [zeros(max(1,size(X,1)),1) ...
0107                 zeros(max(1,size(X,1)),1)];
0108
0109     case 'r(x,t)' %nx1
0110         out1 = ones(max(1,size(X,1)),1);
```

```
0111
0112     case 'dr(x,t)/dt' %nx1
0113         out1 = zeros(max(1,size(X,1)),1);
0114
0115     case 'Lambda' %scalar
0116         out1 = 1;
0117
0118     case 'Beta' %scalar
0119         out1 = 1;
0120
0121     case 'TimeInterval'%1x2 vector
0122         out1 = [0 1];
0123
0124     case 'OptionFile' %specify options file
0125         out1 = 'options';
0126
0127 otherwise
0128         error('+++ Unknown flag');
0129 end
0130
0131 %end of Model1Driver
```

Index

Bibliography

[1] M. AINSWORTH and J.-T. ODEN. *A Posteriori Error Estimation in Finite Element Analysis*. Wiley Intersience, New York, 2000.

[2] P.-W. ATKINS. *Physical Chemistry*. Oxford University Press, Oxford, 5^{th} edition, 1994.

[3] I. BABUŠKA and A.K. AZIZ. Survey lectures on the mathematical foundations of the finite element method. *Academic Press*, pages 3–363, 1972. in "The Mathematical Foundation of the Finite Element Method with Applications to Partial Differential Equations".

[4] M. BEBENDORF. A note on the poincaré inequality for convex domains. *Journal for Analysis and its Applications*, 22(4):751–756, 2003.

[5] D. BRAESS. *Finite Elemente*. Springer, Berlin, 3^{rd} edition, 2003.

[6] JOHNSON C. Adaptive finite element methods for parabolic problems I: A linear model problem. *SIAM J. Numer.Anal.*, 28:43–77, 1991.

[7] C. CARSTENSEN. *Finite Elemente*. lecture notes, CAU Kiel, 2003.

[8] Ph. CIARLET. *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam, 1978.

[9] P. CLÉMENT. Approximation by finite element functions using local regularization. *RAIRO Anal. Numér.*, 9(R–2):77–84, 1975.

[10] K. ERIKSSON and JOHNSON C. *Adaptive Computational Methods for Parabolic Problems*. Encyclopedia of Computational Mechanics Chapter 24. John Wiley & Sons, New York, 2004.

[11] L.-C. EVANS. *Partial Differential Equations*. Oxford University Press, Oxford, 1998.

[12] G. HÖFINGER and JUDEX F. Pollution in groundwater flow – definition of ARGESIM comparison C19. *Simulation News Europe*, 44/45:51–52, 2005.

[13] D. PRAETORIUS. *Einf. in die Numerik Partieller Differentialgleichungen*. UT Vienna, 2005. lecture notes.

[14] W. RITZ. Über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik. *J. reine angew. Math.*, 1908.

[15] R. VERFÜRTH. Robust a posteriori error estimators for a singularly perturbed reaction-diffusion equation. *Numer. Math.*, 78:479–493, 1998.

[16] R. VERFÜRTH. A posteriori error estimates for linear parabolic problems. *Preprint, Ruhr Universität Bochum*, 2004. `http://www.ruhr-uni-bochum.de/num1/rv/papers/APEELPE.pdf` .

[17] R. VERFÜRTH. *Numerische Behandlung von Differentialgleichungen I.* Ruhr Universität Bochum, 2005/2006. lecture notes, `http://www.ruhr-uni-bochum.de/num1/skripten/Numerik_DGL1_Ver2005.pdf`.

[18] E. ZEIDLER. *Taschenbuch der Mathematik*. Teubner, Stuttgart, 2003.