Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Diploma Thesis

hp-DGFEM in Concepts 1.4

Philipp Frauenfelder

March 6, 2000

instructed by

Prof. Christoph Schwab
Seminar for Applied Mathematics
Department of Mathematics
Swiss Federal Institute of Technology
Zürich

Abstract

Discontinuous Galerkin Finite Element Methods (DGFEM) were introduced over 25 years ago for the numerical solution of first-order hyperbolic problems. Although, subsequently much of the research of partial differential equations has concentrated on the development and the analysis of conforming Finite Element Methods (FEM), recent years have witnessed renewed interest in discontinuous schemes. In contrast to standard FEM, DGFEM allow discontinuous numerical solutions. The DGFEM can also be thought of as the higher-order extension of the classical cell centre Finite Volume Method (FVM)—a popular discretization technique in the computational aerodynamics community.

This Diploma thesis is devoted to the implementation of the *hp*-version of the DGFEM as presented in [17]. The motivation is to have a numerical evidence for the proof of exponential convergence in a polygon [18]. The implemented second-order partial differential equation covers a large class of equations which includes advection-dominated diffusion problems and problems of elliptic type. The proof of the exponential convergence does not include the advection term, tough.

Unfortunately, there is an important difference between the implemented DGFEM from [17] and the one used in [18]: the former uses weakly enforced Dirichlet boundary conditions and the latter strongly enforced Dirichlet boundary conditions.

The outline of the thesis is as follows: chapter 1 gives a very short introduction to FEM and then to DGFEM including the variational formulation of the implemented equation. In chapter 2, a summary of the results in [18] is given (the proofs are omitted). Chapter 3 presents the basis of the software which was used (Concepts by Dr. Christian Lage). In chapter 4, the needed extensions to Concepts are explained. And in chapter 5, the numerical results are presented. In the appendix, some of the source code is given for the experienced reader.

The closing remarks can be found on page 61.

Contents

1 Introduction to Discontinuous Galerkin Finite Element Methods

The convergence of any numerical method is based upon *consistency of the approximation* and upon *stability of the discretization.* It is well known that *hp*-FEM can achieve exponential approximation rates for typical second order partial differential equations for certain problems (e. g. in solid mechanics). This requires the combination and simultaneous variation of the polynomial degree and mesh-refinement.

For strongly advection dominated problems, as they appear in computational fluid dynamics, the usual Galerkin type discretisations (as in FEM) do not exhibit good stability properties. Of particular interest are discontinuous approximations which can be used for first order or strongly advection dominated problems. The discontinuous Galerkin FEM (DGFEM) implement such a discontinuous approximation.

This chapter gives first a very brief introduction to standard FEM. Most of the principles found there can also be applied to DGFEM, which are presented in the second section. The last section shows a more sophisticated variational formulation for DGFEM (the one which was used to implement the *hp*-DGFEM).

Most of the information in this chapter is taken from [5] and [17].

1.1 Continuous Discretization

Let $\Omega \subset \mathbb{R}^2$ be a bounded domain with $\Gamma = \partial\Omega$ and $\overline{\Gamma} = \overline{\Gamma}_D \cup \overline{\Gamma}_N$ the partition of Γ into a Neumann and Dirichlet boundary. Consider the model problem

$$\begin{aligned} -\Delta u + u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma_D \\ \frac{\partial u}{\partial n} &= g_N \quad \text{on } \Gamma_N. \end{aligned} \tag{1.1}$$

1.1.1 Variational Form

We introduce the space

$$H_0^1(\Omega) = \left\{ u \in H^1(\Omega) : u = 0 \text{ on } \Gamma_D \right\}.$$

The standard procedure of multiplying with a test function and integrating by parts leads to the variational formulation of (1.1):

$$\text{Find } u \in H_0^1(\Omega) \text{ such that} \int_\Omega \nabla u \cdot \nabla v + uv \, dx = \int_\Omega fv \, dx + \int_{\Gamma_N} g_N v \, ds \quad \forall v \in H_0^1(\Omega). \tag{1.2}$$

This variational formulation is discretised by restricting u and v to a FE-subspace of $H_0^1(\Omega)$.

1.1.2 Finite Element Space

We consider a partition (FE-mesh) $\mathcal{T}$ of Ω into open elements K such that $\bigcup_{K\in\mathcal{T}} \overline{K} = \overline{\Omega}$. We will assume that the $K \in \mathcal{T}$ are images of the reference element $\hat{\Omega} = (0,1)^2$ under affine maps F_K, i. e.:

$$\forall K \in \mathcal{T} : K = F_K(\hat{\Omega}).$$

With each $K \in \mathcal{T}$ we associate a polynomial degree $p_K \geq 1$ and collect them in the degree vector $\underline{p} = \{p_K : K \in \mathcal{T}\}$. $\mathcal{P}_p(\hat{\Omega})$ denotes the polynomials of total degree p on $\hat{\Omega}$.

In order to discretise (1.2), subspaces of continuous functions must be chosen. Usually, one chooses

$$\mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T}) := \left\{ u \in H_0^1(\Omega) : u|_K \circ F_K \in \mathcal{P}_{p_K}(\hat{\Omega}), \forall K \in \mathcal{T} \right\} \subset H_0^1(\Omega),$$

We introduce the bilinear form

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v + uv \, dx \tag{1.3}$$

and the linear form

$$l(v) := \int_\Omega fv \, dx + \int_{\Gamma_N} g_N v \, ds \tag{1.4}$$

and finally discretise (1.2):

$$\text{Find } u_{FE} \in \mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T}) \text{ such that } a(u_{FE}, v) = l(v) \quad \forall v \in \mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T}). \tag{1.5}$$

1.1.3 Linear System

By expressing u_{FE} and v in a basis of $\mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T})$, (1.5) leads to a linear system which can be solved by standard techniques. Let $N := \dim \mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T})$ and $\{\varphi_i\}_{i=1}^N$ be a basis of $\mathcal{S}_0^{\underline{p},1}(\Omega,\mathcal{T})$. Then we can write

$$u_{FE} = \sum_{i=1}^N u_i \varphi_i \text{ and } v = \sum_{j=1}^N v_j \varphi_j.$$

Therefore, (1.5) leads to

$$\begin{aligned} a\left(\sum_{i=1}^{N} u_i\varphi_i, \sum_{j=1}^{N} v_j\varphi_j\right) &= l\left(\sum_{j=1}^{N} v_j\varphi_j\right) \\ \sum_{i,j=1}^{N} u_i a(\varphi_i,\varphi_j) v_j &= \sum_{j=1}^{N} l(\varphi_j) v_j \\ \underline{u}^\top A \underline{v} &= \underline{l}^\top \underline{v} \end{aligned}$$

where

$$A = \{a(\varphi_i,\varphi_j)\}_{i,j=1}^{N}, \underline{u} = \{u_i\}_{i=1}^{N}, \underline{v} = \{v_j\}_{j=1}^{N} \text{ and } \underline{l} = \{l(\varphi_j)\}_{j=1}^{N}.$$

Solving

$$A^\top \underline{u} = \underline{l} \tag{1.6}$$

for $\underline{u}$ results in $u_{FE} = \sum_{i=1}^{N} u_i\varphi_i$ satisfying (1.5).

Remark 1.7 *The presented continuous discretization of* (1.1) *leads to a symmetric A as the bilinear form a in* (1.3) *is symmetric. This won't be the case anymore with the discontinuous discretization. Moreover, A is positive definite:*

$$a(u,u) = \int_\Omega |\nabla u|^2 + u^2\,dx > 0 \textit{ for } u \neq 0 \textit{ in the } L^2\textit{-sense.}$$

1.2 Discontinuous Discretization

The continuity of the FE-solution in the previous section is restrictive. As mentioned in the first paragraphs of this chapter, it is desirable to admit discontinuous approximations for u_{FE}. To this end, the variational formulation (1.2) must be changed. This deserves a new name: Discontinuous Galerkin Finite Element Method, short: DGFEM. We won't treat a detailed derivation of the new variational formulation, see [5] for this. But we will show a heuristic interpretation of the new variational formulation. Later in the chapter we show consistency of a more sophisticated formulation.

We assume that the elements in the subdivision $\mathcal{T}$ are numbered in a certain way. We denote by $\mathcal{E}$ the set of element edges associated with the mesh $\mathcal{T}$. Since hanging nodes are permitted in the DGFEM, $\mathcal{E}$ will be understood to consist of the smallest edges in ∂K.

The changed variational formulation:

$$\text{Find } u_{DG} \in \mathcal{S}_0^{p,0}(\Omega,\mathcal{T}) \text{ such that } B_{DG}(u_{DG}, v) = l(v) \quad \forall v \in \mathcal{S}_0^{p,0}(\Omega,\mathcal{T}), \tag{1.8}$$

where

$$\begin{aligned} B_{DG}(u_{DG}, v) = &\sum_{K\in\mathcal{T}} \int_K \nabla u_{DG} \cdot \nabla v + u_{DG} v \, dx \\ &+ \sum_{e\in\mathcal{E}} \int_e [u_{DG}] \left\langle \nabla v \cdot \underline{n}_e \right\rangle - [v] \left\langle \nabla u_{DG} \cdot \underline{n}_e \right\rangle \, ds, \end{aligned} \tag{1.9}$$

$$\mathcal{S}_0^{p,0}(\Omega, \mathcal{T}) = \left\{ u \in L^2(\Omega) : u|_K \circ F_K \in \mathcal{P}_{p_K}(\hat{\Omega}),\, u|_{\partial K \cap \Gamma_D} = 0, \forall K \in \mathcal{T} \right\}$$

and $l(v)$ the same as in (1.4). From now on, we write $\mathcal{S}_0^p(\Omega, \mathcal{T})$ instead of $\mathcal{S}_0^{p,0}(\Omega, \mathcal{T})$. With the numbering of $\mathcal{T}$ we define

$$[v] := v|_{\partial K_i \cap e} - v|_{\partial K_j \cap e} \tag{1.10}$$

and

$$\langle v \rangle := 1/2 \left(v|_{\partial K_i \cap e} + v|_{\partial K_j \cap e} \right), \tag{1.11}$$

where K_i and K_j share the edge e with $i > j$. $\underline{n}_e$ is defined as the unit normal vector of the edge e pointing from K_i to K_j.

Remark 1.12 *The choice of the minus sign in* (1.9) *is somewhat arbitrary but has a good reason: if we changed it to a plus sign the resulting matrix would be indefinite. With the minus, the matrix is non-symmetric put positive semidefinite:*

$$\forall u \in H^1(\Omega) : B_{DG}(u, u) = \sum_K \int_K |\nabla u|^2 \, dx \geq 0.$$

(1.8) and (1.9) are similar to (1.2)—apart from the different spaces—only the sum over the internal edges in (1.9) is new. As there is no need for continuity anymore, we can choose the basis functions of $\mathcal{S}_0^p(\Omega, \mathcal{T})$ to have support in exactly one element of $\mathcal{T}$. If we evaluate only the first sum in (1.9) with such a basis, the associated matrix has a block diagonal structure, each block resulting from one $K \in \mathcal{T}$. If we would solve this system, an element would not have any connection to its neighbours. This connection is introduced by the second sum in (1.9).

1.3 Advection Diffusion Problem

In the introductory paragraph, advection dominated problems were mentioned. We now consider an advection diffusion problem and show the variational formulation of such a problem. For a more detailed description, see [17].

Let Ω be a bounded Lipschitz domain in $\mathbb{R}^d$, $d = 2$ or 3, and consider

$$-\nabla\big(a(x)\nabla u\big) + \underline{b}(x) \cdot \nabla u + c(x)u = f(x), \tag{1.13}$$

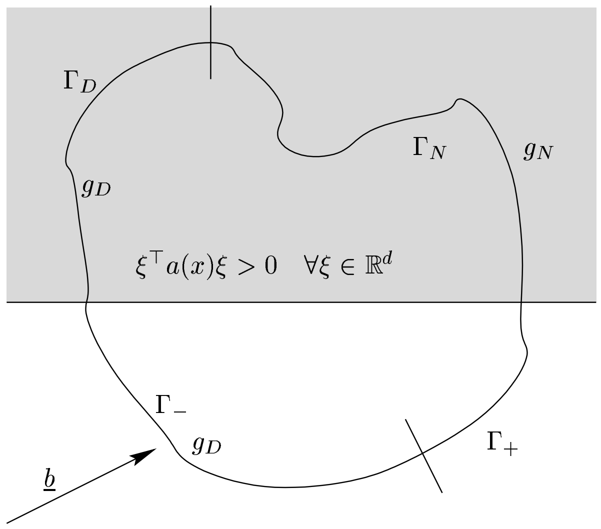


Figure 1.1: Interesting subsets of the boundary.

where $a \in L^\infty(\Omega)^{d\times d}_{\text{sym}}$, $\underline{b}(x) \in W^{1,\infty}(\Omega)^d$, $c(x) \in L^\infty(\Omega)$ and $f(x) \in L^2(\Omega)$. We assume that the principal part of the partial differential operator in (1.13) is nonnegative, i. e.

$$\xi^\top a(x)\xi \geq 0 \quad \forall \xi \in \mathbb{R}^d \text{ and a. e. in } \Omega.$$

Let $\underline{n}$ denote the unit outward normal vector of $\Gamma = \partial\Omega$ and define the following subsets of Γ (see figure 1.1):

$$\begin{aligned} \Gamma_0 &= \left\{x \in \Gamma : \underline{n}^\top a(x)\underline{n} > 0\right\} \text{ diffusion boundary,} \\ \Gamma_- &= \{x \in \Gamma \setminus \Gamma_0 : \underline{b} \cdot \underline{n} < 0\} \text{ inflow boundary,} \\ \Gamma_+ &= \{x \in \Gamma \setminus \Gamma_0 : \underline{b} \cdot \underline{n} \geq 0\} \text{ outflow boundary.} \end{aligned}$$

With these definitions, we have $\Gamma = \Gamma_0 \cup \Gamma_- \cup \Gamma_+$. We further decompose the diffusion boundary Γ_0 into two connected parts: Γ_D, where Dirichlet boundary conditions are imposed, and Γ_N, where Neumann boundary conditions are imposed. Therefore, we can supplement (1.13) with the following boundary conditions:

$$\begin{aligned} u &= g_D \text{ on } \Gamma_D \cup \Gamma_-, \\ \underline{n}^\top a \nabla u &= g_N \text{ on } \Gamma_N. \end{aligned} \tag{1.14}$$

Remark 1.15 *With* (1.13) *and* (1.14) *a wide range of physically relevant problems can be described: a mixed boundary value problem for an elliptic equation if we choose $a(x)$ positive on the whole domain or a linear transport problem with the choice $a(x) \equiv 0$ on $\overline{\Omega}$.*

1.3.1 Variational Form

The variational form of (1.13) needs to allow for the possibility of $a = 0$ on the boundary. Additionally, the Dirichlet boundary conditions are weakly enforced, i. e. they are part of the variational formulation and not of the DGFE-space.

The variational formulation is split into four parts: the diffusion term, the advection term, the absolute term and a discontinuity penalisation term.

Diffusion Term

We have already introduced $\mathcal{E}$, the set of edges of the elements $K \in \mathcal{T}$. In addition, we define the set of interior element edges

$$\mathcal{E}_{\text{int}} := \{e \in \mathcal{E} : e \cap \partial\Omega = \emptyset\}$$

and its union

$$\Gamma_{\text{int}} := \bigcup_{e \in \mathcal{E}_{\text{int}}} e.$$

The variational formulation of the left and right hand side are:

$$\begin{aligned} B_a(u,v) &= \sum_{K \in \mathcal{T}} \int_K \nabla u \cdot a \nabla v \, dx + \int_{\Gamma_D} u\big((a\nabla v) \cdot \underline{n}_e\big) - \big((a\nabla u) \cdot \underline{n}_e\big) v \, ds && (1.16) \\ &\quad + \int_{\Gamma_{\text{int}}} [u] \left\langle (a\nabla v) \cdot \underline{n}_e \right\rangle - \left\langle (a\nabla u) \cdot \underline{n}_e \right\rangle [v] \, ds \\ l_a(v) &= \int_{\Gamma_D} g_D \big((a\nabla v) \cdot \underline{n}_e\big) \, ds + \int_{\Gamma_N} g_N v \, ds && (1.17) \end{aligned}$$

Advection Term

Similarly to the inflow boundary of the whole domain, we define the inflow boundary of an element $K \in \mathcal{T}$:

$$\partial_{-}K := \{x \in \partial K : \underline{b} \cdot \underline{n}_K < 0\},$$

where $\underline{n}_K$ denotes the unit outward normal vector on K. In contrast to the numbering dependent jump $[v]$, we introduce the oriented jump of v over an edge $e \subset \partial K \setminus \Gamma$:

$$\lfloor v \rfloor := v^{+} - v^{-},$$

where v^{+} is the inner trace of v in K. The definition on $\partial K \setminus \Gamma$ implies that there exists an element K' sharing e with K. Therefore we can define the outer trace v^{-} of v on e relative to K as the inner trace of v on e relative to K'.

Remark 1.18 *In general, $[v]$ will be distinct from $\lfloor v \rfloor$ as the former depends on the numbering and the latter does not. However, $|[v]| = |\lfloor v \rfloor|$.*

$$B_b(u,v) = \sum_{K\in\mathcal{T}} \int_K (\underline{b}\cdot\nabla u)v\,dx - \sum_{K\in\mathcal{T}} \int_{\partial_- K\setminus\Gamma_-} (\underline{b}\cdot\underline{n}_K)\lfloor u \rfloor v^+\,ds \qquad (1.19)$$

$$- \sum_{K\in\mathcal{T}} \int_{\partial_- K\cap\Gamma_-} (\underline{b}\cdot\underline{n}_K)u^+v^+\,ds$$

$$l_b(v) = -\sum_{K\in\mathcal{T}} \int_{\partial_- K\cap\Gamma_-} (\underline{b}\cdot\underline{n}_K)g_D v^+\,ds \qquad (1.20)$$

Absolute Term

$$B_c(u,v) = \sum_{K\in\mathcal{T}} \int_K cuv\,dx \qquad (1.21)$$

$$l_c(u,v) = \sum_{K\in\mathcal{T}} \int_K fv\,dx \qquad (1.22)$$

Discontinuity Penalisation Term

We define $\delta_K := h_K^{-1}$, where $h_K := \operatorname{diam} K$, i. e. is element wise constant. δ_K is a stabilisation parameter.

$$B_s(u,v) = \int_{\Gamma_D} \delta_K uv\,ds + \int_{\Gamma_{\text{int}}} \delta_K[u][v]\,ds \qquad (1.23)$$

$$l_s(v) = \int_{\Gamma_D} \delta_K g_D v\,ds \qquad (1.24)$$

Remark 1.25 *The proof of the exponential convergence, see section 2.4, relies on this stabilisation.*

The numerics in chapter 5 show that the stabilisation is not necessary for the chosen model problem.

The variational formulation of (1.13) is then

$$B_{DG}(u,v) := B_a(u,v) + B_b(u,v) + B_c(u,v) + B_s(u,v) = l_a(v) + l_b(v) + l_c(v) + l_s(v) =: l_{DG}(v). \qquad (1.26)$$

1.3.2 Consistency

To show consistency, we plug the solution of (1.13) and (1.14) into (1.26) and assume the solution to be differentiable and continuous. Therefore, $[v] = \lfloor v \rfloor = 0$ and $\langle v \rangle = v$.

We start with treating the diffusion term (1.16) with integration by parts:

$$\begin{aligned} B_a(u,v) &= \int_\Omega \nabla u \cdot a \nabla v \, dx + \int_{\Gamma_D} u\big((a\nabla v)\cdot \underline{n}\big) - \big((a\nabla u)\cdot \underline{n}\big) v \, ds \\ &= \int_\Omega -\nabla(a\nabla u) v \, dx + \int_{\Gamma_D} u\big((a\nabla v)\cdot \underline{n}\big)\, ds + \int_{\Gamma_N} \big((a\nabla u)\cdot \underline{n}\big) v \, ds, \end{aligned}$$

since a is symmetric. Together with $l_a(v)$ from (1.17), the second and third term weakly fulfill the boundary condition on Γ_D and Γ_N. What remains is $\int_\Omega -\nabla(a\nabla u)v\,dx$.

Next, we treat the advection term (1.19):

$$B_b(u,v) = \int_\Omega (\underline{b}\cdot\nabla u) v \, dx - \int_{\Gamma_-} (\underline{b}\cdot\underline{n}) uv \, ds.$$

There is nothing to do since $l_b(v)$ from (1.20) and the second term weakly fulfill the boundary condition on Γ_-. What remains is $\int_\Omega (\underline{b}\cdot\nabla u)v\,dx$.

For $B_c(u,v)$ and $l_c(v)$ from (1.21) and (1.22) respectively, there is nothing to do.

The last term is the discontinuity penalisation term (1.23):

$$B_s(u,v) = \int_{\Gamma_D} \delta_K uv \, ds.$$

Together with $l_s(v)$ from (1.24), $B_s(u,v)$ weakly fulfills the boundary condition on Γ_D.

We sum up all the terms which remained:

$$\int_\Omega -\nabla(a\nabla u)v\,dx + \int_\Omega (\underline{b}\cdot\nabla u)v\,dx + \int_\Omega cuv\,dx$$

on the left hand side and

$$\int_\Omega fv\,dx$$

on the right hand side, i. e.

$$\int_\Omega \left[-\nabla(a\nabla u) + \underline{b}\cdot\nabla u + cu - f\right] v\,dx = 0.$$

We therefore conclude consistency of (1.26) with (1.13) and (1.14).

2 Exponential Convergence in DGFEM

The main result of this chapter is the exponential convergence of hp-DGFEM on a polygon. The model problem presented herein features homogeneous Dirichlet boundary conditions which are strongly enforced in the variational formulation (in contrast to the variational formulation presented in section 1.3.1 and implemented in chapter 4).

Essentially, this chapter is a summary of [18] which provides the theoretical proof of what the computations in chapter 5 show.

2.1 The Model Problem and its Regularity

Let $\Omega \subset \mathbb{R}^2$ be a bounded, polygonal domain. We assume that its boundary $\Gamma = \partial\Omega$ is composed of a Dirichlet part Γ_D with $\int_{\Gamma_D} ds > 0$ and of a Neumann part Γ_N: $\overline{\Gamma} = \overline{\Gamma}_D \cup \overline{\Gamma}_N$.

We consider the model problem

$$\begin{aligned} -\nabla(a\nabla u) + cu &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma_D \\ (a\nabla u)\cdot \underline{n} &= g_N \quad \text{on } \Gamma_N. \end{aligned} \tag{2.1}$$

Here, the coefficients $a(x)$ and $c(x)$ have the following properties:

$$a(x) = \{a_{ij}(x)\}_{i,j=1}^{2} \in W^{1,\infty}(\Omega)^{2\times 2}_{\text{sym}} \text{ and } c(x) \in L^\infty(\Omega). \tag{2.2}$$

Further we assume that $c(x) \geq 0$ for all $x \in \Omega$ and (2.1) is properly elliptic, i. e.:

$$\exists \bar{a}, a_0 > 0 : \bar{a}\,|\xi|^2 \geq \xi^\top a(x)\xi \geq a_0\,|\xi|^2 \quad \forall \xi \in \mathbb{R}^2, x \in \overline{\Omega}. \tag{2.3}$$

We will measure the regularity of (2.1)–(2.3) in terms of certain weighted Sobolev spaces: Let $\Omega \subset \mathbb{R}^2$ be a polygonal domain and let A_i, $i = 1, \ldots, M$ denote its vertices. Further let $\beta = (\beta_1, \ldots, \beta_M)$, $0 \leq \beta_i < 1$, be an M-tuple. For any number k we define $\beta \pm k := (\beta_1 \pm k, \ldots, \beta_M \pm k)$.

We define the **weight function** $\Phi_\beta(x)$ by

$$\Phi_\beta(x) := \sum_{i=1}^{M} r_i^*(x)^{\beta_i} \quad \text{where } r_i^*(x) := \min\{1, |x - A_i|\}.$$

For integers $0 \leq l \leq m$ we introduce the semi-norms

$$|u|^2_{H^{m,l}_\beta(\Omega)} := \sum_{k=l}^{m} \left\| |D^k u| \, \Phi_{\beta+k-l} \right\|^2_{L^2(\Omega)} .$$

By $H^{m,l}_\beta(\Omega)$, $0 \leq l \leq m$, we denote the completion of $\mathcal{C}^\infty(\overline{\Omega})$ with respect to the norms

$$\begin{aligned} \|u\|^2_{H^{m,l}_\beta(\Omega)} &:= \|u\|^2_{H^{l-1}(\Omega)} + |u|^2_{H^{m,l}_\beta(\Omega)} \quad \text{for } l \geq 1, \\ \|u\|^2_{H^{m}_\beta(\Omega)} &:= \sum_{k=0}^{m} \left\| |D^k u| \, \Phi_{\beta+k} \right\|^2_{L^2(\Omega)} \quad \text{for } l = 0. \end{aligned}$$

Definition 2.4 (Countably Normed Space $\mathcal{B}^l_\beta(\Omega)$) *Fix $l \geq 0$ and M-tuple $\beta = (\beta_1, \ldots, \beta_M)$. The countably normed space $\mathcal{B}^l_\beta(\Omega)$ consists of all functions u for which $u \in H^{m,l}_\beta(\Omega)$ for all $m \geq l$ and*

$$\left\| |D^k u| \, \Phi_{\beta+k-l} \right\|_{L^2(\Omega)} \leq C d^{k-l} (k-l)! \quad \textit{for } k = l, l+1, \ldots$$

for some constants $C > 0$ and $d \geq 1$ independent of k.

We also need certain trace spaces of $\mathcal{B}^l_\beta(\Omega)$. To this end, let $\mathcal{M} \subset \{1, \ldots, N\}$ be an index set and define

$$\gamma = \bigcup_{j \in \mathcal{M}} \overline{\Gamma}_j \subset \Gamma.$$

Then $\mathcal{B}^{l-1/2}_\beta(\Omega)$ is the set of traces from $\mathcal{B}^l_\beta(\Omega)$ on γ.

Now, for the solution of problem (2.1)–(2.3) we have the following

Theorem 2.5 (Regularity) *Let $\Omega \subset \mathbb{R}^2$ be a polygon. Then there exist $0 < \beta_j < 1$, $j = 1, \ldots, N$, such that for $f \in \mathcal{B}^0_\beta(\Omega)$ and $g_N \in \mathcal{B}^{1/2}_\beta(\Omega)$ the solution of problem (2.1)–(2.3) exists and belongs to $\mathcal{B}^2_\beta(\Omega)$.*

Proof: See [3] and [4]. □

2.2 *hp*-DGFEM

We now introduce the finite element spaces. Note that in this theoretical part (as in [18]) the homogeneous Dirichlet boundary conditions are strongly enforced. For convenience, in chapters 4 and 5, boundary conditions are weakly enforced, see section 1.3.1.

2.2.1 Variational Formulation

The variational formulation can be taken from section 1.3.1 if we take into account that the Dirichlet boundary conditions are strongly enforced and that $\underline{b} = 0$:

$$\begin{aligned} B(u,v) = & \sum_{K\in\mathcal{T}} \int_K \nabla u \cdot a(x)\nabla v + c(x)uv\,dx \\ & + \int_{\Gamma_{\text{int}}} [u]\left\langle (a\nabla v)\cdot \underline{n}_e\right\rangle - \left\langle (a\nabla u)\cdot \underline{n}_e\right\rangle [v]\,ds \\ & + \sum_{K\in\mathcal{T}} \delta_K \int_{\partial K\setminus\Gamma} [u][v]\,ds \end{aligned} \tag{2.6}$$

$$l(v) = \sum_{K\in\mathcal{T}} \int_K fv\,dx + \int_{\Gamma_N} g_N v\,ds. \tag{2.7}$$

We define the energy norm:

$$|u|_{DG}^2 := B(u,u) = \Big(\sum_{K\in\mathcal{T}} \left\|\sqrt{a}\nabla u\right\|_{L^2(K)}^2 + \left\|\sqrt{c}u\right\|_{L^2(K)}^2 + \delta_K \left\|[u]\right\|_{L^2(\partial K\setminus\Gamma)}^2\Big) \tag{2.8}$$

and

$$\|u\|_{DG}^2 := \Big(\sum_{K\in\mathcal{T}} \|u\|_{H^1(K)}^2 + h_K^{-1}\|u\|_{L^2(\partial K\setminus\Gamma)}^2 + h_K |u|_{H^1(\partial K\setminus\Gamma)}^2\Big)$$

Remark 2.9 *The proofs of the stability and the exponential convergence need the stabilisation term in* (2.6). *In the numerical part (in chapter 5), we will see that the stabilisation is not needed for the exponential convergence on the chosen model problem.*

2.3 Stability

In the following section, some result about the stability and convergence of the *hp*-DGFEM are presented.

Let $\pi_p u_{\text{ex}} \in \mathcal{S}_0^{\underline{p}}(\Omega,\mathcal{T})$ be an arbitrary interpolant of the solution u_{ex} of our model problem (2.1)–(2.3). Further, let u_{DG} denote the solution of the DGFEM defined by the bilinear form (2.6) and the linear form (2.7). Then we write

$$u_{\text{ex}} - u_{DG} = \underbrace{u_{\text{ex}} - \pi_{\underline{p}} u}_{=:\eta} + \underbrace{\pi_{\underline{p}} u - u_{DG}}_{=:\xi}. \tag{2.10}$$

In the ensuing Proposition 2.11 we will prove that ξ may be estimated by η. Hence, we will be able to bound the error $u_{\text{ex}} - u_{DG}$ of the *hp*-DGFEM by η only.

Proposition 2.11 (Stability) *Let $\mathcal{G} = \{\mathcal{T}_i\}_{i\in\mathbb{N}}$ be a shape regular mesh family. Further let the conditions* (2.2) *and* (2.3) *be be satisfied.*

Then, there exists a constant $C > 0$ depending only on the constants in (2.2) *and* (2.3) *and on the regularity of the mesh such that for the hp-DGFEM defined by* (2.6) *and* (2.7) *with $\delta_K = h_K^{-1}$ there holds the a-priori estimate*

$$|\xi|_{DG} \leq C p_{i,\max} \|\eta\|_{DG} \quad \forall i \in \mathbb{N},$$

where $p_{i,\max} := \max_{K\in\mathcal{T}_i} p_K$.

Proof: See [18]. □

2.4 Convergence

From Proposition 2.11 we know that the error $|u_{\text{ex}} - u_{DG}|_{DG}$ of the *hp*-DGFEM may be estimated by $\eta = u_{\text{ex}} - \pi_{\underline{p}} u_{\text{ex}}$ in a certain way, where $\pi_{\underline{p}} u_{\text{ex}} \in \mathcal{S}_0^{\underline{p}}(\Omega, \mathcal{T})$ may be arbitrarily chosen. Therefore, we are interested in *hp*-approximations of the exact solution u_{ex} of the model problem (2.1)–(2.3).

We will prove that by a judicious combination of mesh refinement towards the singular points of the polygon (i. e. corner vertices and vertices with changing boundary condition type) and increase of the polynomial degree p used in the approximation, exponential convergence may be achieved. To do so, we will introduce the so-called *geometric meshes* on Ω.

Elliptic regularity states that the solution u_{ex} is, for analytic data, itself analytic in $\overline{\Omega}$ minus these singularities. Geometrically graded meshes therefore ensure the analyticity of the solution restricted to each element not abutting at a singular point.

2.4.1 Approximation on the Unit Square

Definition 2.12 (Geometric Mesh Family $\hat{\mathcal{T}}_\sigma^n$) *On $\hat{\Omega} = (0,1)^2$ we define the geometric mesh family $\hat{\mathcal{T}}_\sigma^n$ with $n+1$ layers and grading factor $0 < \sigma < 1$ recursively as follows: if $n = 0$, $\hat{\mathcal{T}}_\sigma^0 = \{\hat{\Omega}\}$. Given $\hat{\mathcal{T}}_\sigma^n$, $n \geq 0$, generate $\hat{\mathcal{T}}_\sigma^{n+1}$ by subdividing the element K_0 abutting at the singular vertex $0 \in \overline{K_0}$ into three smaller rectangles by diving all but one of its sides in a $\sigma : (1-\sigma)$ ratio (c. f. Figure 2.1).*

Proposition 2.13 (Approximation on $\hat{\Omega}$) *Let $\hat{\Omega} = (0,1)^2$, $0 < \sigma < 1$ and let $\hat{\mathcal{T}}_\sigma^n$ be the geometric mesh with $n+1$ layers defined in Definition 2.12. Then, for $u \in \mathcal{B}_\beta^2(\hat{\Omega})$ with some $0 < \beta < 1$ and $\Phi_\beta(x) = |x|^\beta$, there exists $\mu > 0$ such that for the following linear polynomial degree distribution vector with slope μ*

$$\underline{p} = \left\{p_{ij} = p_i, i = 0, \ldots, n, j = 1, 2, p_i = \max\left\{1, \lfloor \mu i \rfloor\right\}\right\}$$

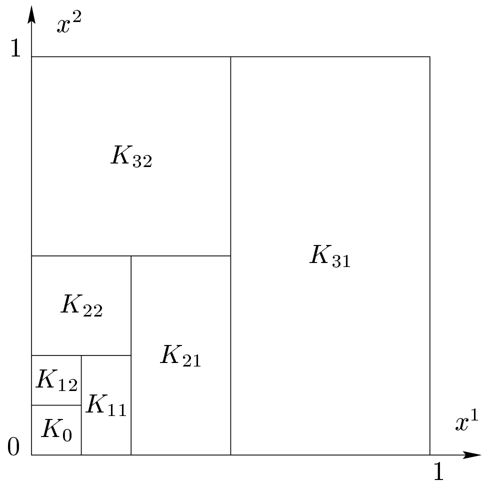


Figure 2.1: Geometric mesh $\hat{\mathcal{T}}_\sigma^n$ for $\sigma = 0.5$ and $n = 3$.

there holds:

$$\inf_{v \in \mathcal{S}_0^{\underline{p}}(\Omega, \mathcal{T}_\sigma^n)} \|u - v\|_{DG} \leq C e^{-b\sqrt[3]{N}},$$

where $C, b > 0$ *do not depend on* $N = \dim \mathcal{S}_0^{\underline{p}}(\hat{\Omega}, \hat{\mathcal{T}}_\sigma^n)$.

2.4.2 Approximation on a Polygon

We will now establish an exponential convergence result like Proposition 2.13 on a polygon Ω. The basic idea will be to localize the *hp*-approximation problem at each singular point and to use the results in the previous section.

Definition 2.14 (Geometric Meshes on a Polygon) *A geometric mesh family* $\mathcal{T}_\sigma^n$ *on a polygon* Ω *is obtained by mapping the basic mesh* $\hat{\mathcal{T}}_\sigma^n$ *from* $\hat{\Omega}$ *linearly to a vicinity of each convex corner of* Ω. *At reentrant corners, three and at Dirichlet/Neumann vertices two copies of* $\hat{\mathcal{T}}_\sigma^n$ *(suitably scaled), are used, see figure 2.2. The remaining domain* $\tilde{\Omega}$ *is meshed with a fixed mesh which is regularly connected with the geometric patches at the singular points.*

In order to construct a piecewise *hp*-approximation $\pi_{\underline{p}} u \in \mathcal{S}_0^{\underline{p}}(\Omega, \mathcal{T}_\sigma^n)$ of $u \in \mathcal{B}_\beta^2(\Omega)$, we proceed as follows: first we construct $\pi_{\underline{p}} u$ in each parallelogram patch. Then, since u is analytic in $\tilde{\Omega}$, we increase the polynomial degree on the fixed mesh in $\tilde{\Omega}$ consistently with the largest degree in each patch (we assume that the degree vectors in each geometric mesh patch are identical), yielding exponential convergence also in $\tilde{\Omega}$. Summing up the local error estimates from the subregions leads to the ensuing result:

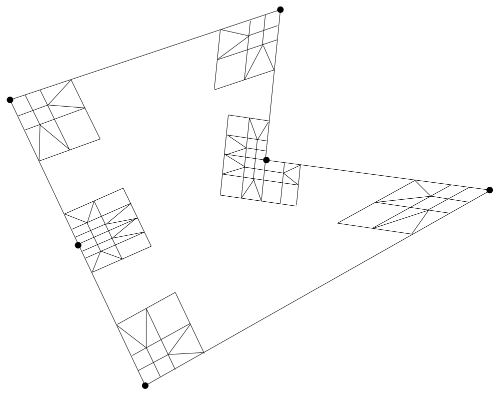


Figure 2.2: Polygon with geometric meshes in the singular vertices.

Proposition 2.15 *Let $\Omega \subset \mathbb{R}^2$ be a polygon and $u \in \mathcal{B}^2_\beta(\Omega)$.*
Then there holds

$$inf_{v \in \mathcal{S}^p_0(\Omega, \mathcal{T}^n_\sigma)} \|u - v\|_{DG} \leq C e^{-b \sqrt[3]{N}},$$

where $C, b > 0$ are independent of $N = \dim \mathcal{S}^p_0(\Omega, \mathcal{T}^n_\sigma)$.

Proof: In order to prove this proposition, a generalization of Proposition 2.13 to parallelograms is needed. This can be established similarly to Proposition 2.13. □

2.4.3 Convergence on a Polygon

The main result of [18] is

Theorem 2.16 *Let $\Omega \subset \mathbb{R}^2$ be a polygon and let moreover the conditions* (2.2) *and* (2.3) *be satisfied. Then, for the hp-DGFEM defined by* (2.6) *and* (2.7) *on $\mathcal{S}^p_0(\Omega, \mathcal{T}^n_\sigma)$ with $\delta_K = h_K^{-1}$ there holds the following error estimate:*

$$|u_{\text{ex}} - u_{DG}|_{DG} \leq C e^{-b \sqrt[3]{N}},$$

where $C, b > 0$ are independent of $N = \dim \mathcal{S}^p_0(\Omega, \mathcal{T}^n_\sigma)$.

Proof: See [18]. □

3 Introduction to Concepts

Concepts is a library nearly completely written in C++, an object oriented programming language [16]. The main benefits of C++ are:

- One can implement very efficient code.
- Compilers are available on almost all platforms [1] and C++ is one of the most widespread object oriented programming languages.
- With the availability of templates, name spaces and exceptions, in addition to the Standard Template Library [9], reusable code on a rather high level can be written.

In [11], Lage describes the main design idea behind the design of Concepts as follows:

> Since we are interested in the development of numerical software, there is a special situation: the considered numerical methods are already formulated in an abstract way based on hierarchical structured mathematical concepts. This motivates the following approach: represent each concept by a module and combine these modules according to the numerical algorithm to generate an implementation.

With this in mind, it is easier to understand the design of Concepts.

In this chapter, the main parts of Concepts are presented. Starting with the second section, the design principles following the mathematical hierarchy of FEM are explained. To support this, only class hierarchy diagrams are used. To better understand the interactions of the presented classes, the fourth section shows some UML diagrams. The fifth section explains the element integration of the Laplacian in great detail. The first section gives a short overview of the history and the authors of Concepts.

3.1 History and Authors

The software was mainly written by *Dr. Christian Lage.* First versions and ideas of the software appear already in his Ph. D. thesis [10]. The current version 1.4 was developed during his post doctoral studies at the Seminar for Applied Mathematics of the Swiss Federal Institute of Technology (ETH), Zurich. The design ideas leading the implementation of Concepts are presented in [11].

Ana-Maria Matache is working with the *hp*-FEM part of Concepts. She implemented the quadrilaterals for two dimensional problems together with Lage. She is the one I have asked if I did not know what happened in the code. She also gave me the first introductions to Concepts.

Recently (summer 1999), two students (*David Hoch* and *Andreas Rüegg*) wrote a semester thesis [15] on and with Concepts-1.4. They implemented mixed and variable boundary conditions for *hp*-FEM. Their work was supervised by Ana-Maria. They both continue to work on Concepts and have already implemented an interface to a direct sparse solver.

3.2 Main Parts of Concepts

The library Concepts implements the concepts of both Finite Element Methods and Boundary Element Methods (BEM). In the following, only the FEM part is treated. Much of the principles are applicable to BEM too, tough.

Typography

The following typographical conventions are used:

- Class names are typeset like class name.
- Abstract classes are typeset like ***abstract***. An abstract class is merely an interface class, i. e. not all methods have an implementation, they only serve to prescribe the interface of the derived classes.

Mathematical Concepts and their Classes

The quotation of Lage above suggests we have a closer look at the concepts of FEM. We do so by repeating (1.5), the discretised formulation of FEM:

$$\text{Find } u_{FE} \in \mathcal{S}_0^{p,1}(\Omega, \mathcal{T}) \text{ such that } a(u_{FE}, v) = l(v) \quad \forall v \in \mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$$

and (1.6), the linear system resulting from (1.5): $A^\top \underline{u} = \underline{l}$. Therefore, we need concepts for:

1. The mesh $\mathcal{T}$ on the domain Ω.
2. The shape functions on $\hat{\Omega}$ and the element maps F_K leading to a basis $\{\varphi_i\}_{i=1}^N$ of the FE-space $\mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$.
3. The bilinear form a and the linear form l leading to the matrix A and the vector $\underline{l}$.
4. A solver for the linear system.

These concepts are implemented in the following classes:

1. The domain of interest Ω is directly described by the mesh in a user defined class, e. g. geo_Quadrat for the unit square. This class is derived from *geo_Mesh2* in the two dimensional case.

2. The space is implemented in hp_HP2d. Like the definition of $\mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$ suggests, it consists of elements hp_HP2d001 (quadrilaterals). For the sake of efficiency, the elements directly incorporate the shape functions on the reference element. Therefore, the implementation can exploit the special structure of the shape functions.

3. hp_Laplace and hp_Identity are both derived from *op_BilinearForm*. Together, they form the implementation of a. hp_Riesz for l is derived from *fnc_LinearForm*.

   op_Operator is the base class for all operators. Hence, op_Matrix for A is derived from it. The class for $\underline{l}$ is called fnc_Vector.

4. The inverse of an operator can be computed by different means: op_GMRes implements the General Minimal Residuals algorithm and op_CG the Conjugate Gradients algorithm, op_DGESV is an interface to a direct solver from LAPACK [2]. They are all derived from *op_Operator*.

In the rest of the section, we will look at the different mathematical concepts and the according classes.

3.2.1 Topology, Geometry and Mesh

Topology

The topology of a mesh is described by means of connectors. These connectors are vertices, edges, faces (in three dimensions) and cells.

Restricted to the case of two space dimensions, a cell consists of a number of edges—four in the case of a quadrilateral and three in the case of a triangle. Furthermore, an edge consists of two vertices. This is reflected in the classes of the geometry package. The hierarchy of the mentioned classes is shown in figure 3.1.

The class *geo_Connector* prescribes the common interface for all topological elements (connectors). This interface consists of a method to query the attribute of the connector. A typical application for the attribute of a connector are boundary conditions. Additional interface methods are prescribed on the next level of derivation. The refinement of the elements of the topology—i. e. the subdivision—is implemented on the highest level of derivation, i. e. in geo_Edge etc.

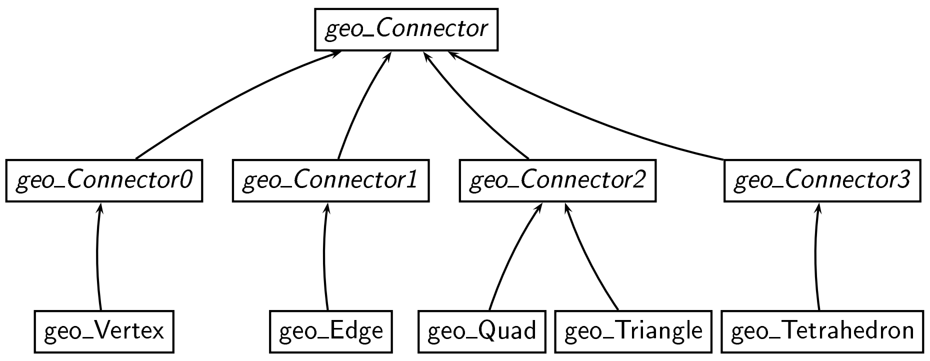


Figure 3.1: The class hierarchy in the topological part of the geometry package. An arrow represents a "is a" relation.

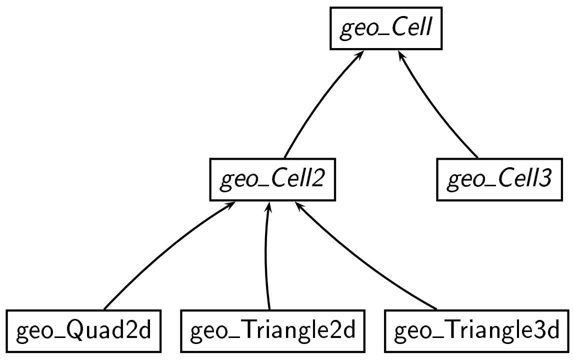


Figure 3.2: The class hierarchy in the geometrical part of the geometry package.

Geometry

Until now, we have only seen the topology of the cells in a meshed domain. With the concept of geometry, we introduce the notion of coordinates by adding the element mapping to the cells of the topology.

A cell of the topology, e. g. geo_Quad, together with the element mapping of this cell results in a quadrilateral of the geometry, e. g. geo_Quad2d. As one would expect, the classes on the highest level of derivation have a method to evaluate the element mapping, i. e. map a point in the reference element onto a physical point. The hierarchy of the abovementioned classes is shown in figure 3.2.

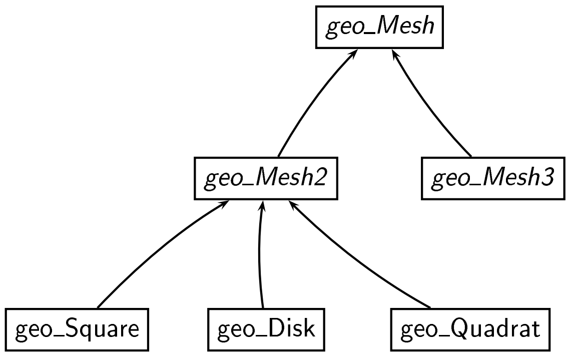


Figure 3.3: The class hierarchy in the mesh part of the geometry package. The classes geo_Square, geo_Disk and geo_Quadrat implement examples of meshes. They are sketched in figure 3.4.

Mesh

The topology and geometry as introduced above do not describe a mesh. They only describe separate cells in a mesh. The classes in the mesh part of the geometry package pack the cells together and finally give an implementation of the mathematical notation $\mathcal{T}$. The classes in this part of the geometry package are shown in figure 3.3.

Usually, the class implementing the mesh has to be user defined. Some examples like geo_Square, geo_Disk and geo_Quadrat are already available, c. f. figure 3.4.

3.2.2 Shape Functions, Elements and Space

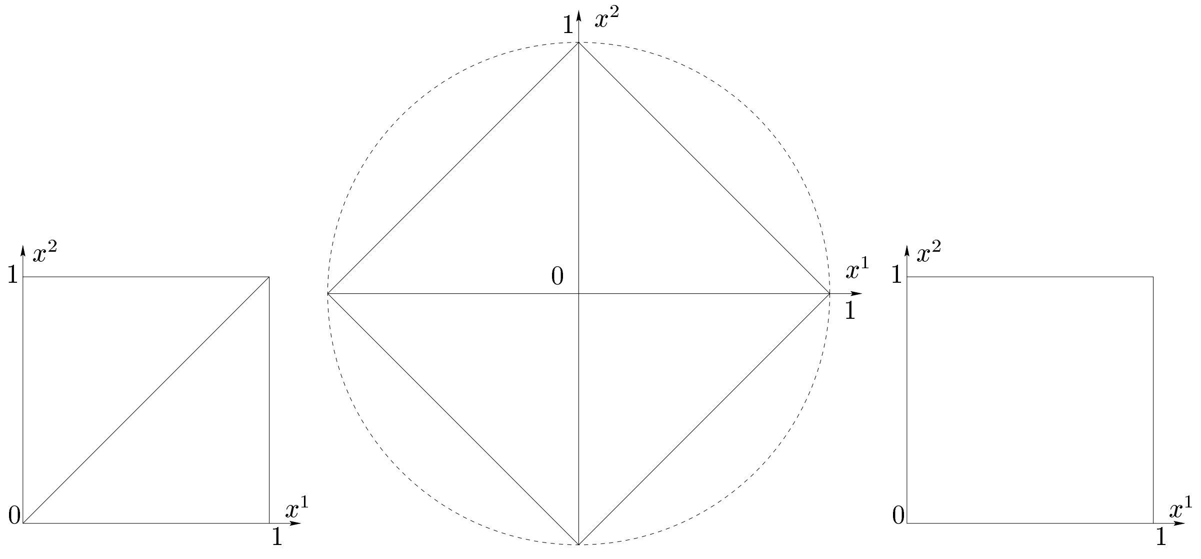
Shape Functions

As stated above, the shape functions are integrated into the class for the elements of the FE-space. The advantages of this Ansatz are that one can exploit the special structure of the chosen shape functions. The disadvantage: the classes for the elements cannot be reused if one wants to exchange the shape functions.

Elements

An element of the FE-space consists of the support of the element (including the coordinates) and of the polynomial degree on this element. In two dimensions, the support of an element is a derived class of *geo_Cell2*, c. f. figure 3.2.

The abstract class for an element of the FE-space *hp_HP2dXXX* (c.f. figure 3.5) prescribes the interface to query the polynomial degree and the support of an element. As stated in the paragraph about shape functions above, these are also included into the element. Therefore, the classes hp_HP2d000 and hp_HP2d001 both have methods to evaluate all the necessary shape functions at a specific point in the reference element.



(a) geo_Square (b) geo_Disk (c) geo_Quadrat

Figure 3.4: The meshes implemented by the example classes. geo_Square is the unit square $(0,1)^2$ meshed with two triangles. geo_Disk is the unit disk $\{x \in \mathbb{R}^2 : |x| < 1\}$ initially meshed with four triangles—the mesh does not really resemble a disk but refining the mesh will improve the situation: the new vertices on the boundary will lie on the circle. geo_Quadrat meshes the unit square $(0,1)^2$ with one quadrilateral.

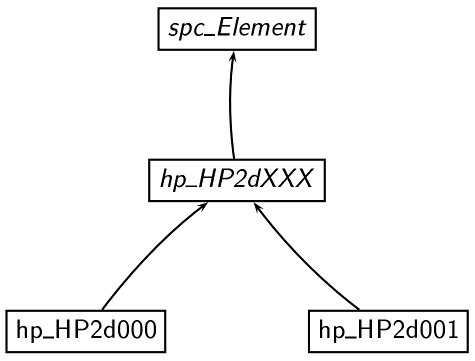


Figure 3.5: The class hierarchy in the element part of the space package. hp_HP2d000 implements triangular elements, hp_HP2d001 is used for quadrilaterals.

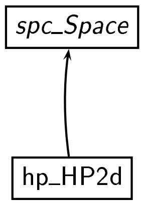


Figure 3.6: Class hierarchy in the space part of the space package.

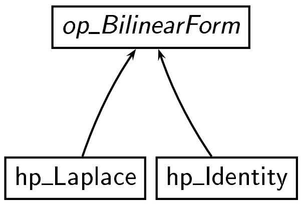


Figure 3.7: Class hierarchy in the bilinear forms part of the operator package.

The topology describes how the elements are connected. In FEM, we need to have a continuous basis of the FE-space. Hence, there exist certain conditions which have to be fulfilled at inter-element boundaries. From the algorithmic point of view, this is achieved by globally enumerating the degrees of freedom and then mapping the local degrees of freedom of each element onto the global ones. According to this mapping, the global matrix is assembled from the element matrices. This mapping is stored element wise in the so-called T matrices.

The interface to access these matrices is prescribed on the level of *spc_Element*.

Space

As the mesh in the geometry package collects all the cells in the topology and their geometrical information to implement the mesh $\mathcal{T}$, the FE-space packs the above-described elements together and implements $\mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$.

The abstract class *spc_Space* prescribes the interface to query the dimension and the number of elements of the space and to get an iterator (a so-called scanner) of the space. With such a scanner one can iterate over all elements of the space (in no particular order, tough).

The class hp_HP2d, derived from *spc_Space* (c.f. figure 3.6), implements these mandatory methods. Additionally, there are methods to set refinement requirements on the elements of the space and to finally refine and rebuild the space. During this process, the T matrices of the elements are computed and stored.

3.2.3 Bilinear Forms and Operators

Bilinear Forms

A bilinear form is evaluated for two functions of the basis of $\mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$, the result is stored in the global matrix: $A = \{a(\varphi_i, \varphi_j)\}_{i,j=1}^{N}$. Algorithmically, it is evaluated for all shape functions on all elements to form the element matrices.

The abstract function class *op_BilinearForm* prescribes the interface to evaluate a bilinear form on two elements, i. e. to compute $A_{KK'} = \left\{a(\varphi_i^K, \varphi_j^{K'})\right\}$ where $i = 1, \ldots, N_K$ and $j = 1, \ldots, N_{K'}$ are the indices of the shape functions on the elements K and K'

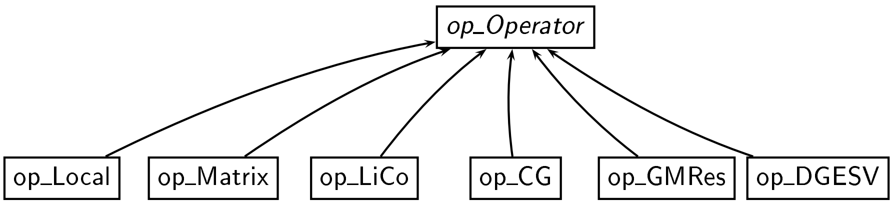


Figure 3.8: Class hierarchy in the matrices and solver part of the operator package.

respectively. A function class is a class whose main usage is through the parenthesis operator (called application operator). *op_BilinearForm* prescribes the interface for the application operator.

The derived class hp_Laplace (c. f. figure 3.7) implements an application operator according to the interface prescribed by *op_BilinearForm* which evaluates the variational form of the Laplacian, i. e. $\int_K \nabla u \cdot \nabla v \, dx$ for the shape functions u and v on element K:

$$A_K = \left\{a(\varphi_i^K, \varphi_j^K)\right\}_{i,j=1}^{N_K},$$

where $\left\{\varphi_i^K\right\}_{i=1}^{N_K}$ are the shape functions on element K.

Similarly to hp_Laplace, hp_Identity evaluates $\int_K uv \, dx$ on a given element K.

Operators

To form the global matrix, the element matrices have to be computed and then assembled into the global matrix. This is done in a op_Local or op_Matrix class, c. f. figure 3.8. The abstract base class *op_Operator* prescribes the interface for an operator consisting of an application operator which computes the application of an operator on a vector.

The class op_Matrix implements a dense matrix and op_Local a matrix in sparse notation: only the non-zero entries have to be stored. op_LiCo stands for a linear combination of two operators.

3.2.4 Linear Forms and Vectors

Linear Forms

A linear form $l(v)$ is evaluated for each basis function of $\mathcal{S}_0^{p,1}(\Omega, \mathcal{T})$ and the result is stored in the load vector. As for bilinear forms, algorithmically, it is evaluated for all shape functions on all elements to form the element load vectors.

The abstract function class *fnc_LinearForm* (see figure 3.9) prescribes the interface of a linear form: the application operator computes and returns the local load vector for a

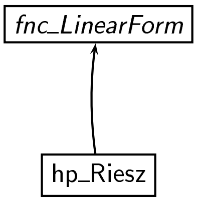


Figure 3.9: Class hierarchy in the linear forms part of the function package.

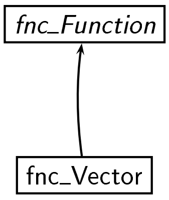


Figure 3.10: Class hierarchy in the vector part of the function package.

given element of the space:

$$l_K = \left\{ l(\varphi_j^K) \right\}_{j=1}^{N_K}.$$

The element load vector is returned as a $N_K \times 1$ matrix.

Vectors

The vector class fnc_Vector, c. f. figure 3.10, is declared similarly to the class for matrices (see the section about operators above). The element load vectors are computed using a given linear form and are then assembled into the global load vector.

The biggest difference to the matrix classes op_Matrix and op_Local: as the vectors are not stored in sparse notation, they support a wide variety of operations (such as addition, subtraction and scaling).

3.2.5 Solver

Solving a linear system is nothing more than computing the inverse of the matrix and applying the right hand side to it. Therefore, a solver is an operator which takes a matrix and a vector and calculates the result.

There are several different solvers implemented: Conjugate Gradients in op_CG, General Minimal Residuals in op_GMRes or a direct solver from LAPACK [2] in op_DGESV. As these are all operators, they are derived from *op_Operator*, see figure 3.8.

3.3 Main Steps in a Concepts Application

The main steps in a Concepts application are:

1. Create the mesh.
2. Create the space.
3. (Refine the mesh.)

4. Create the bilinear forms and the according matrices and combine them to form the stiffness matrix.
5. Create the linear forms and the according vectors and combine them to form the load vector.
6. Create a Solver and solve the linear system.
7. (Analyse the error and proceed with step 3 if the error is too large.)
8. Post processing: plots, error computation etc.
9. Remove the matrices, vectors, the space and the mesh.

The steps in parenthesis are optional.

3.4 Some UML Diagrams of Concepts

Until now, we have only seen class hierarchies. They don't present very much information about the interactions of the classes. In most object oriented codes, these interactions are very complex—Concepts is no exception.

The graphical representation of complex correlations gives a much faster and better overview than a description in a text. Since this is not really new, the *Unified Modeling Language* (UML) was developed [12, 13, 14] and [7] for a short introduction. It defines nine different diagrams for the graphical description of object oriented designs.

I have chosen three of those nine diagrams as suitable to describe the structure of Concepts: the *Class*, *Object* and *Sequence Diagram*.

3.4.1 Class Diagram

The Class Diagram describes states of a data structure in a general form characterizing a set of allowed states. To achieve this, classes and their member relations are shown. An arrow in a Class Diagram describes a *member relation*, i. e. the class at the tail of an arrow has as member a reference to an object of the class at the head of the arrow. The label of the arrow is the name of the member object. The number on the head counts the number of references which are needed by the "tail class", in most cases they don't need to be distinct, e. g. spaceX and spaceY in figure 3.11 may be identical. The number on the tail counts the number of classes which have a reference to the same object (n means "arbitrary").

The Class Diagram in figure 3.11 shows the main classes which take part in computing the stiffness matrix and the load vector for a Laplace equation on the unit square $(0, 1)^2$ meshed with geo_Quadrat (see figure 3.4(c) on page 20).

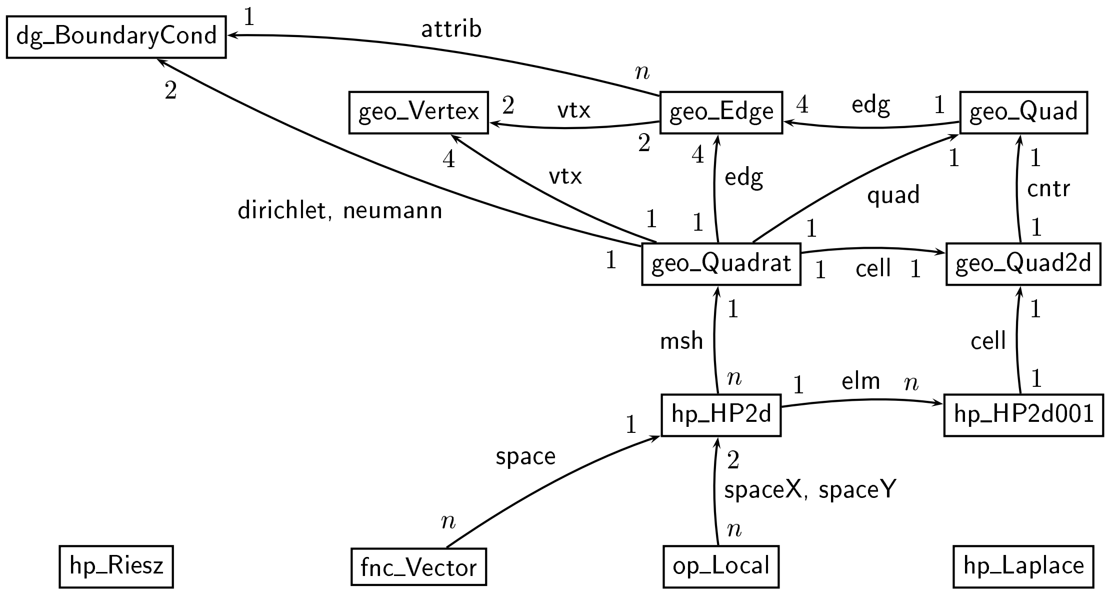


Figure 3.11: Class Diagram of vector, matrix, space and topology. The bilinear form hp_Laplace and the linear form hp_Riesz are only used temporarily by op_Local and fnc_Vector respectively. Arrows which would describe a loop, i. e. have the same head and tail, are not shown.

3.4.2 Object Diagram

Like the Class Diagram, the Object Diagram describes states of a data structure. Not in general tough, but for one concrete state, i. e. like a snapshot of the structure. To achieve this, the objects and their member relations are shown. An arrow in a Object Diagram describes a *member relation*, i. e. the object at the tail of an arrow has as member a reference to the object at the head of the arrow.

The most complicated part of the structure in Concepts is the geometry and topology, above all, when the mesh is refined. Figure 3.12 shows as the initial mesh the unit square $(0, 1)^2$ and then the refinement of the square into four smaller squares. Figures 3.13 and 3.14 show the data structure of the initial and the refined mesh respectively.

Initial Mesh

In figure 3.13, from left to right are the arrays for the vertices (type geo_Vertex), the edges (type geo_Edge), the quadrilateral (of the topology, type geo_Quad) and the cell (type geo_Quad2d). One can easily see that each edge has two references to vertices and the quadrilateral has four references to edges (see also the Class Diagram in figure 3.11). The arrow on the left crossing many others indicates that the edges are arranged circularly.

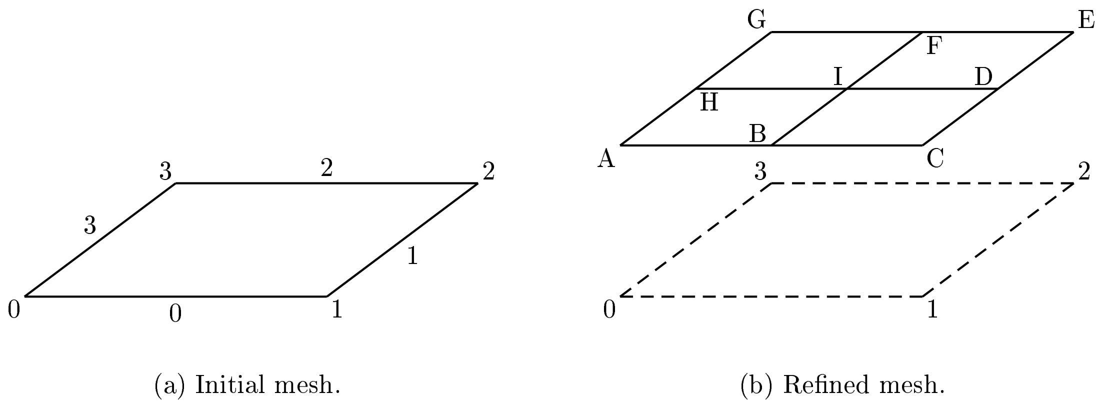


Figure 3.12: The initial mesh and the refined mesh. The numbers indicate the indices of the vertices and the edges in the vtx and edg array respectively. Topologically, the vertices 0 and A are identical, but in the data structure, they are two different objects.

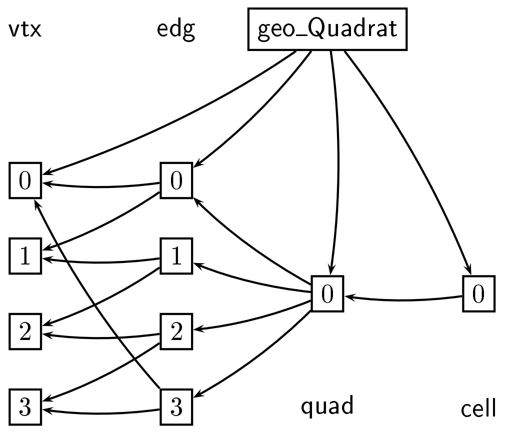


Figure 3.13: The Object Diagram of the initial mesh (c. f. figure 3.12(a)). The numbers indicate the indices of the entries in the vtx, edg, quad and cell arrays of the mesh object (symbolized by geo_Quadrat).

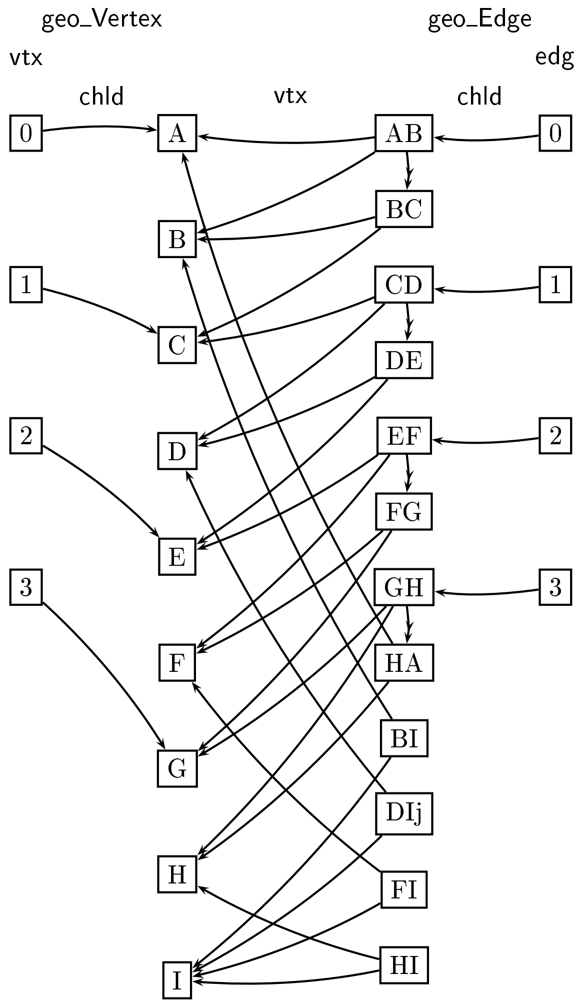


Figure 3.14: The Object Diagram of the refined mesh (c.f. figure 3.12(b)). Only the objects of type geo_Vertex (on left) and geo_Edge (on the right) are displayed. The numbered objects are the same as in figure 3.13. The members of geo_Edge indicated by $\twoheadrightarrow$ are called lnk.

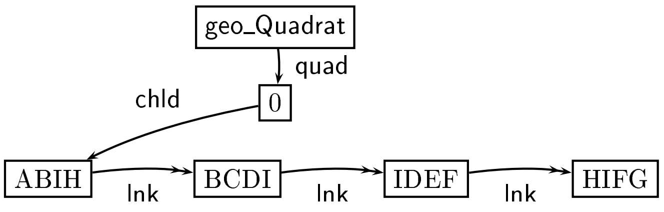


Figure 3.15: Object diagram of the children of the geo_Quad object in the refined mesh: all objects except the one labeled geo_Quadrat are of the type geo_Quad.

Refined Mesh

The data structure of the refined mesh (c. f. figure 3.12(b)) is already quite complicated. Therefore, only parts of it (only the objects of type geo_Vertex and geo_Edge) are shown in figure 3.14.

Each object which was already present in the initial mesh has got one or more children in the refined mesh. A vertex has only one child, an edge has two children and a quadrilateral has four children (see figure 3.12(b)). This is also visible in figure 3.14: the geo_Vertex objects on the left have one chld arrow pointing to a letter-labeled vertex. The geo_Edge objects on the left also have a chld arrow point to a letter-labeled edge. This letter-labeled edge has an additional lnk arrow. The children of a topological object are not stored in an array but in a linked list. For the children of the geo_Quad object this would look like in figure 3.15.

3.4.3 Sequence Diagram

A Sequence Diagram pictures the interaction among objects. It shows the participating objects together with a life line (the long rectangle beneath the object's name). The messages the objects exchange (symbolized by arrows) are ordered on the life line with respect to their occurrence in time. The numbers of the phases correspond to the numbers in section 3.3.

Mesh Creation

Figure 3.16 shows the sequence of major operations to create the mesh (phase 1). The constructor of the mesh class is called from the main program and then all the objects which form the topology (see figure 3.13) are created and properly arranged. The order of creation of these objects is: vertices, edges, quadrilateral, cell. The new call to msh only returns after the new call to cell, i. e. one call from the main program is enough to create the mesh. The same holds for the destruction of the mesh: one call suffices.

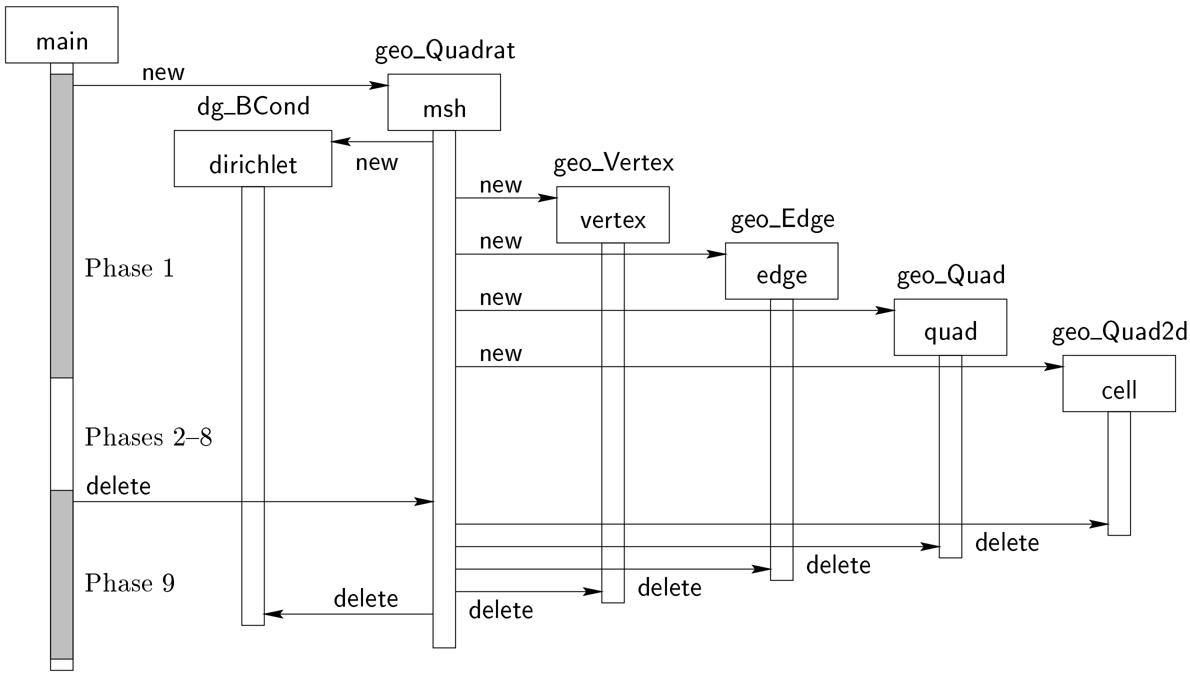


Figure 3.16: Sequence Diagram of the mesh creation: in phase 1, the mesh is created. In the phases 2–8 the calculations are performed (starting with the creation of the space)—this is not shown here. In phase 9, the mesh is removed as one of the last steps in the main program.

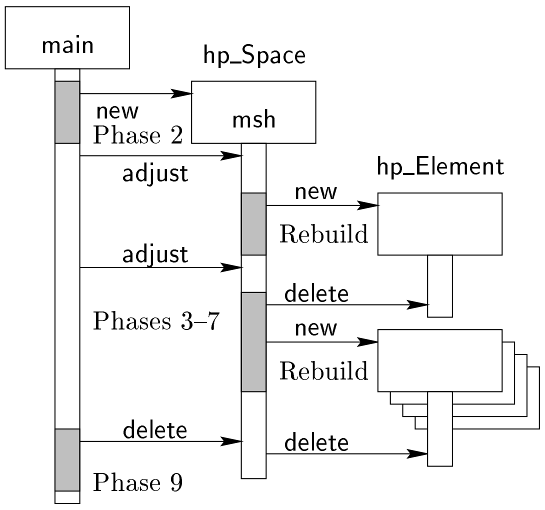


Figure 3.17: Sequence Diagram of the space creation: in phase 2, the space is created. In the phases 3–8 the calculations are performed and the space is refined (once in this example). In phase 9, the space is removed. In this diagram, the rebuild process is triggered twice.

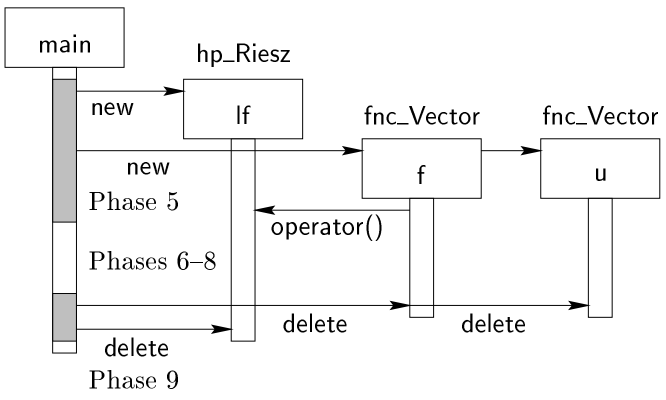


Figure 3.18: Sequence Diagram of the creation of the vectors: in phase 5, the linear forms and the according vectors are created. In the phases 6–8 the calculations are performed—this is not shown here. In phase 9, the linear form and the vectors are removed.

Creation of the Space

Figure 3.17 shows the sequence of operations to create the space (phase 2). The main parameter of the constructor of the space is the mesh which was created in phase 1. The space is only prepared in the constructor, but the elements of the space are not constructed yet. As long as the elements or some other information like number of elements or dimension is not needed, the elements are not constructed. In this phase, the adjustment information can be set, i. e. which elements should be refined and what polynomial degree they should have.

Then, when the elements are eventually necessary, they are constructed in two steps by the so-called rebuild process. In the first step, the topology is refined if needed and prepared for the second step which creates the elements, counts the degrees of freedom and calculates the mapping from the local to the global degrees of freedom (stored in the T matrices of the elements).

Linear Forms and Vectors

Figure 3.18 shows the sequence of operations to create a linear form, its vector f and the solution vector u (phase 5).

Bilinear Forms and Matrices, Solve

Figure 3.19 shows the sequence of operations to create the stiffness matrices from the bilinear forms (phase 4) and then the solver operator Linv. The application operator operator() of Linv is then called with two vectors: the right hand side and the empty solution vector which is then filled with the solution of the linear system (phase 6).

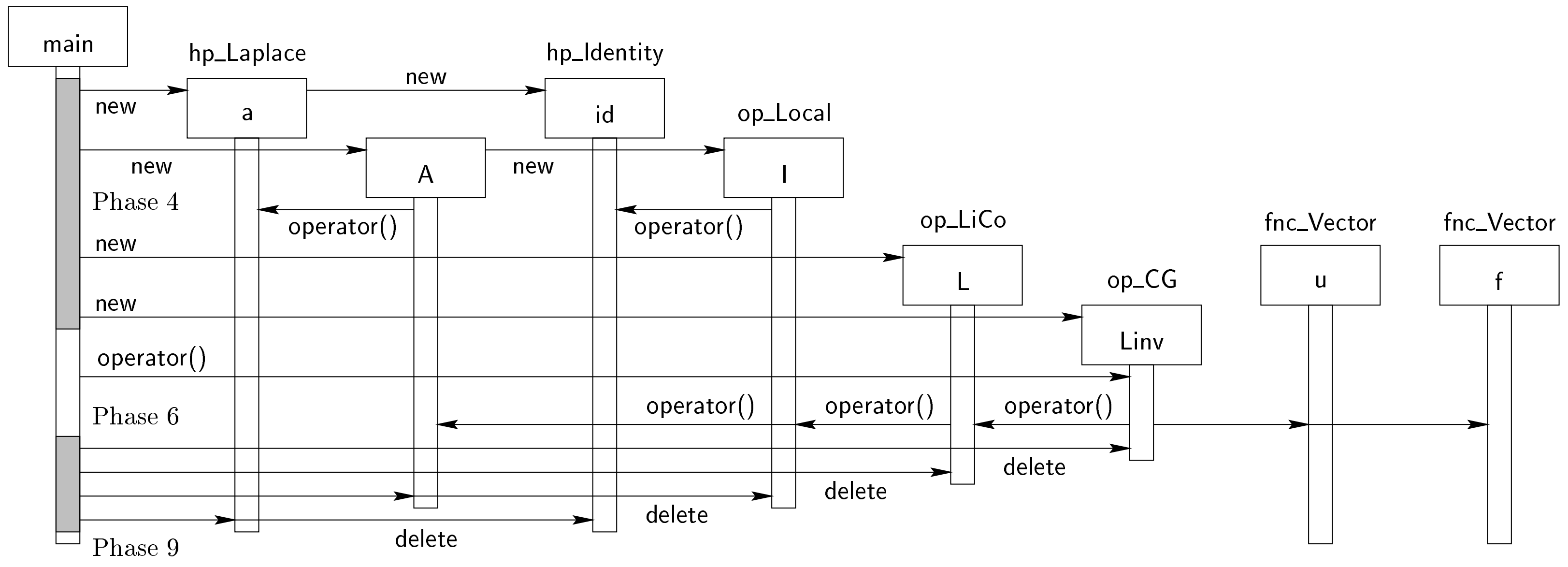


Figure 3.19: Sequence Diagram of the creation of the matrices and the solution: in phase 4, the bilinear forms and the matrices are created. In the phase 6 the solution of the linear system is computed. In phase 9, all data structures are removed. operator() is the C++ notation for the application operator.

3.5 Element Integration

In this section, the element integration algorithm as it is used in the application operator for quadrilaterals of hp_Laplace is discussed in greater detail.

The application operator has to compute

$$\int_K \nabla_x \varphi_i^K \nabla_x \varphi_j^K \, dx = \int_{\hat\Omega} \nabla_x N_i \nabla_x N_j \cdot |F_K'| \, d\xi, \tag{3.1}$$

where ξ are the coordinates in the reference element $\hat\Omega$, $|F_K'|$ is the Jacobian of the element mapping F_K, φ_i^K are the shape functions on element K, N_i are the shape functions on $\hat\Omega$: $\varphi_i^K \circ F_K = N_i$. Therefore,

$$\nabla_x N_i = \begin{pmatrix} \frac{\partial N_i}{\partial x^1} \\ \frac{\partial N_i}{\partial x^2} \end{pmatrix}, \text{ where } \frac{\partial N_i}{\partial x^1} = \frac{\partial N_i}{\partial \xi_1}\frac{\partial \xi_1}{\partial x^1} + \frac{\partial N_i}{\partial \xi_2}\frac{\partial \xi_2}{\partial x^1}. \tag{3.2}$$

In Concepts, currently only bilinear elements are used. Therefore, the element map of element $K \in \mathcal{T}$ is

$$F_K : \hat\Omega \to \Omega, \xi \mapsto \sum_{i=1}^4 x_i \cdot N_i(\xi),$$

where $\{x_i\}_{i=1}^4$ are the vertices of the quadrilateral K. The derivatives of the element mapping are therefore

$$\frac{\partial F_K}{\partial \xi_1} = 1/2\big(-x_1(1-\xi_2) + x_2(1-\xi_2) + x_3\xi_2 - x_4\xi_2\big) = \begin{pmatrix} \frac{\partial x^1}{\partial \xi_1} \\ \frac{\partial x^2}{\partial \xi_1} \end{pmatrix} =: 1/2 s,$$

$$\frac{\partial F_K}{\partial \xi_2} = 1/2\big(-x_1(1-\xi_1) - x_2\xi_1 + x_3\xi_1 + x_4(1-\xi_1)\big) = \begin{pmatrix} \frac{\partial x^1}{\partial \xi_2} \\ \frac{\partial x^2}{\partial \xi_2} \end{pmatrix} =: 1/2 t.$$

Hence, $|F_K'| = \frac{\partial F_K}{\partial \xi_2} \wedge \frac{\partial F_K}{\partial \xi_2} = s \wedge t/4$.

The integrand in (3.1) is (using (3.2))

$$\begin{aligned}
\nabla_x N_i \nabla_x N_j &= \left(\frac{\partial N_i}{\partial \xi_1}\frac{\partial \xi_1}{\partial x^1} + \frac{\partial N_i}{\partial \xi_2}\frac{\partial \xi_2}{\partial x^1}\right) \cdot \left(\frac{\partial N_j}{\partial \xi_1}\frac{\partial \xi_1}{\partial x^1} + \frac{\partial N_j}{\partial \xi_2}\frac{\partial \xi_2}{\partial x^1}\right) \\
&\quad + \left(\frac{\partial N_i}{\partial \xi_1}\frac{\partial \xi_1}{\partial x^2} + \frac{\partial N_i}{\partial \xi_2}\frac{\partial \xi_2}{\partial x^2}\right) \cdot \left(\frac{\partial N_j}{\partial \xi_1}\frac{\partial \xi_1}{\partial x^2} + \frac{\partial N_j}{\partial \xi_2}\frac{\partial \xi_2}{\partial x^2}\right) \\
&= \frac{\partial N_i}{\partial \xi_1}\frac{\partial N_j}{\partial \xi_1} \cdot \left[\left(\frac{\partial \xi_1}{\partial x^1}\right)^2 + \left(\frac{\partial \xi_1}{\partial x^2}\right)^2\right] \\
&\quad + \left(\frac{\partial N_i}{\partial \xi_1}\frac{\partial N_j}{\partial \xi_2} + \frac{\partial N_i}{\partial \xi_2}\frac{\partial N_j}{\partial \xi_1}\right) \cdot \left[\frac{\partial \xi_1}{\partial x^1}\frac{\partial \xi_2}{\partial x^1} + \frac{\partial \xi_1}{\partial x^2}\frac{\partial \xi_2}{\partial x^2}\right] \\
&\quad + \frac{\partial N_i}{\partial \xi_2}\frac{\partial N_j}{\partial \xi_2} \cdot \left[\left(\frac{\partial \xi_2}{\partial x^1}\right)^2 + \left(\frac{\partial \xi_2}{\partial x^2}\right)^2\right].
\end{aligned}$$

We use s, t and $|F'_K|$ to compute the terms in square brackets above:

$$(F'_K)^{-1} = \begin{pmatrix} 1/2s & 1/2t \end{pmatrix}^{-1} = 4/s\wedge t \cdot \begin{pmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{pmatrix} = \begin{pmatrix} \frac{\partial\xi_1}{\partial x^1} & \frac{\partial\xi_1}{\partial x^2} \\ \frac{\partial\xi_2}{\partial x^1} & \frac{\partial\xi_2}{\partial x^2} \end{pmatrix}.$$

Hence

$$\begin{aligned} \left(\frac{\partial\xi_1}{\partial x^1}\right)^2 + \left(\frac{\partial\xi_1}{\partial x^2}\right)^2 &= \frac{t_1^2 + t_2^2}{|F'_K|^2} =: \frac{\mathsf{tt}}{|F'_K|}, \\ \frac{\partial\xi_1}{\partial x^1}\frac{\partial\xi_2}{\partial x^1} + \frac{\partial\xi_1}{\partial x^2}\frac{\partial\xi_2}{\partial x^2} &= -\frac{s \cdot t}{|F'_K|^2} =: \frac{\mathsf{st}}{|F'_K|}, \\ \left(\frac{\partial\xi_2}{\partial x^1}\right)^2 + \left(\frac{\partial\xi_2}{\partial x^2}\right)^2 &= \frac{s_1^2 + s_2^2}{|F'_K|^2} =: \frac{\mathsf{ss}}{|F'_K|}. \end{aligned}$$

Plugging all into (3.1):

$$\int_{\hat\Omega} \left[\frac{\partial N_i}{\partial\xi_1}\frac{\partial N_j}{\partial\xi_1} \cdot \mathsf{tt} + \left(\frac{\partial N_i}{\partial\xi_1}\frac{\partial N_j}{\partial\xi_2} + \frac{\partial N_i}{\partial\xi_2}\frac{\partial N_j}{\partial\xi_1}\right) \cdot \mathsf{st} + \frac{\partial N_i}{\partial\xi_2}\frac{\partial N_j}{\partial\xi_2} \cdot \mathsf{ss}\right] \cdot \frac{|F'_K|}{|F'_K|}\, d\xi.$$

All derivatives to x have vanished and are replaced by derivatives to ξ. These derivatives are defined on the reference element $\hat\Omega$ and no longer on the element K—they are easily computable.

The same method is also applicable to integrate the advection term, where only one of the two shape function has a derivative.

4 Extensions to Concepts for DGFEM

The present chapter describes the extensions of Concepts which were necessary for DG-FEM to work. The first section describes the new classes which implement the DGFEM. In the second section, the really new ideas (new to Concepts), which were introduced, are presented. The third section describes the other extensions which were not really necessary but proved quite useful. The last section gives some ideas what could be done next as it lists possible directions of development.

4.1 New Classes for DGFEM

Most of the extension were done in new classes, directly derived from the base classes or from the classes in the *hp*-FEM package. The figures 4.1–4.6 show these relations.

dg_HP2d001 in figure 4.1 contains nothing new which would be absolutely necessary for DGFEM, but some of the changes made my life as programmer somewhat easier (see section 4.3.2). The changes leading to dg_HP2d (figure 4.2) are described in section 4.2.1. The new class dg_Edge (figure 4.3) was necessary to detect elements sharing an edge, see section 4.2.2. dg_BoundaryCond (figure 4.4) is used to specify non-homogeneous boundary conditions of Dirichlet or Neumann type, see section 4.3.1. The abstract class *dg_BForm* in figure 4.5 provides a few auxiliary computation methods for the derived classes. The classes on the highest level of derivation in figures 4.5 and 4.6 implement the bilinear and linear forms of the variational formulation presented in section 1.3.1.

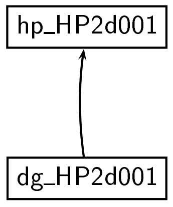


Figure 4.1: dg_HP2d001 is the new quadrilateral element for DGFEM extending the classes shown in figure 3.5.

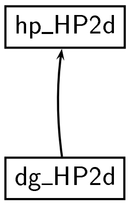


Figure 4.2: The new space class for DGFEM extending the classes shown in figure 3.6: dg_HP2d.

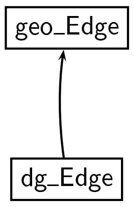


Figure 4.3: The new topological edge dg_Edge for DGFEM extending the classes shown in figure 3.1.

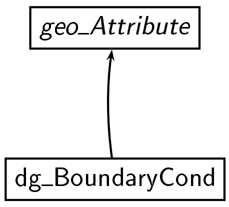


Figure 4.4: The new boundary condition class dg_BoundaryCond.

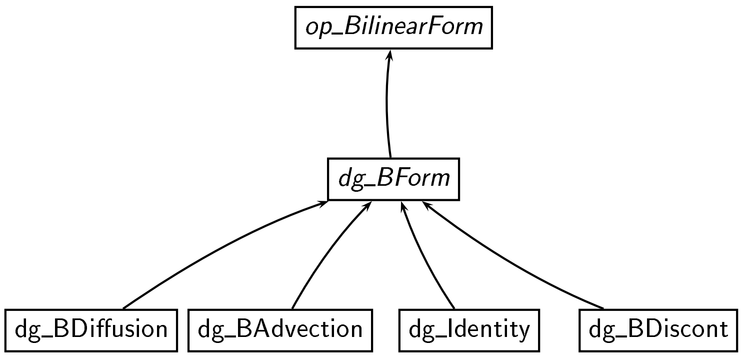


Figure 4.5: The new bilinear forms dg_BDiffusion, dg_BAdvection, dg_Identity and dg_BDiscont implementing the bilinear forms from section 1.3.1.

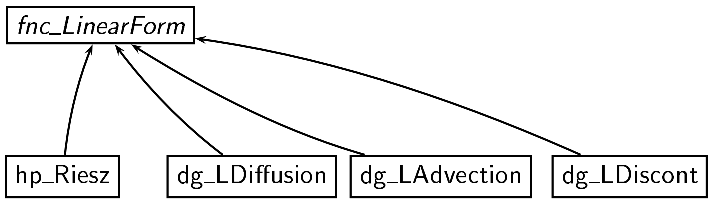


Figure 4.6: The new linear forms dg_LDiffusion, dg_LAdvection and dg_LDiscont and the old hp_Riesz implementing the linear forms from section 1.3.1.

Algorithm 4.1 The controlling method rebuild of the rebuild process.

```
• Remove all old elements and the control information of their edges and vertices from the space.
  The control information of the cells is retained.
• Get a scanner over the initial mesh and for each cell in the initial mesh:
    – Call rebuild0 on level 0 with desired level $-1$ and desired polynomial degree $-1$.
• Get another scanner over the initial mesh and for each cell in the initial mesh:
    – Call rebuild1.
• Remove the adjustment information which was applied during this rebuild process.
```

4.2 New Ideas for Concepts

4.2.1 Discontinuous Elements

When I started my diploma thesis, only FEM were implemented. The FEM use a continuous discretization which is not suitable for DGFEM.

Problem

The continuity of the basis functions of the FE-space is enforced in the rebuild process of the space (c. f. section 3.4.3). So I looked somewhat deeper into this rebuild process.

The whole process is controlled by the method rebuild in the class hp_Space. The sequence of operations is shown in algorithm 4.1. The main thing it does: loop twice over all cells in the initial mesh and call the methods rebuild0 and rebuild1 for each cell.

The method rebuild0 works recursively and determines which cells in the topology should be the support for an element in the space and what their polynomial degree should be. See algorithm 4.2 for the sequence of operations. The level of an element is the level of refinement in the topology. An element in the initial mesh is on level 0. One refinement step on such an element creates four elements on level 1 (c. f. figure 3.12 on page 26).

The method rebuild1 also works recursively. In rebuild1, the degrees of freedom are counted, the T matrices are computed and the elements are created, see algorithm 4.3. The continuity is enforced where the degrees of freedom for the vertices and edges are marked as counted.

Solution

Essentially, the only thing I had to do was remove this marking operation. In rebuild0, the changes are too small to show them with help of algorithm 4.2: on irregular edges, the larger edge is allowed to have a polynomial degree different from the smaller edges.

Algorithm 4.2 Sequence of operations for rebuild0: updating the information in the topology.

```
$l$ = current level, $L$ = desired level, $P$ = desired polynomial degree.
  • If $L = -1$ and this cell belonged to an element in the space:
      – Get the adjustment information for the level $l_{\mathrm{adj}}$ and the polynomial degree $p_{\mathrm{adj}}$.
      – $L = l + l_{\mathrm{adj}}$.
      – $P$ = previous polynomial degree $+p_{\mathrm{adj}}$.
      – If $L$ or $P$ are still negative, then let $L = 0$ or $P = 1$ respectively.
  • If we have to refine the current cell ($l < L$) or if the current cell did not belong to an element in
    the space:
      – For each child of the current cell:
          ▷ Call rebuild0 on level $l + 1$, with desired level $L$ and desired polynomial degree $P$.
        If a child of a cell is accessed for the first time and is not yet present in the topology, the
        access triggers its creation.
  • If we have to refine the current cell ($l < L$):
      – Update the information on the cell and its edges regarding the polynomial degree.
  • otherwise (i. e. no refinement):
      – Update the information on the cell and its edges regarding the polynomial degree.
      – Set the cell, its edges and vertices as “member of the space”, i. e. their degrees of freedom
        will be counted and there will be an element belonging to this cell.
```

Algorithm 4.3 Sequence of operations for rebuild1: count degrees of freedom, compute T matrices and create elements.

```
• Loop over all vertices of the cell:
    – If this vertex is member of the space:
        ▹ If the global degree of freedom of this vertex was not yet counted:
            · Count the degree of freedom of vertex and mark it as counted. The number of
              the global degree of freedom of the vertex is stored in its control information.
        ▹ Add the mapping of the local to the global degree of freedom to the T matrix.
• Loop over all edges of the cell and do the same degree of freedom counting as for the vertices.
• If the cell is member of the space:
    – Count the degrees of freedom and mark them as counted. The degrees of freedom of the
      cell are stored in its control information.
    – Add the mapping of the local to the global degrees of freedom to the T matrix.
    – Create the element for the cell and add it to the space.
• otherwise (i. e. the cell is not member of the space):
    – For each child of the current cell:
        ▹ Recompute the T matrix for the already computed degrees of freedom of the edges
          and vertices.
        ▹ Call rebuild1.
```

Algorithm 4.4 Sequence of operations to assemble a symmetric stiffness matrix.

- Loop over all elements K of the space:
  - Compute A_K by using the application operator of the bilinear form.
  - Get the T matrix T_K of K.
  - Transform the local degrees of freedom into global ones by $\left((A_K T_K)^\top \cdot T_K\right)^\top = T_K^\top A_K T_K$.
  - Add $T_K^\top A_K T_K$ entry by entry into the global stiffness matrix.

In rebuild1 the changes were only minor too: I removed the check if a degree of freedom was already counted. The result is that a vertex is counted once for every element it is in. The same for the edges. Therefore, the shape functions are no longer assembled in such a way that the resulting basis functions are continuous.

4.2.2 Integration over an Edge

Many integrals in the variational formulation in section 1.3.1 are over an edge and not over a whole element like in FEM (section 1.1). This is a new idea in Concepts as until now, all integrals were computed over whole elements and not over an element boundary like an edge.

Problems

More than one problem arises if integration over an edge is required:

1. Not only the shape functions on the same element interact with each other but also those on neighbouring elements have interactions.

   The evaluation of e. g. $[u] \langle (a\nabla v) \cdot \underline{n}_e \rangle$ like in (1.16) on an edge requires the values of u and v from both sides of the edge. The interface of a bilinear form prescribed by op_BilinearForm (section 3.2.3) is such that the method computing the bilinear form gets two elements on which the considered shape functions for u and v have their support. Hence, means to find out if two elements share an edge are needed.

   This is not always very simple since these two elements can be located on different levels of the topology and there are no possibilities to compare them or their edges directly.

2. In the FEM problems solved until now, only symmetric stiffness matrices were computed.

   These were stiffness matrices where the elements interacted only with themselves and not with their neighbours. Therefore, the constructor of op_Local or op_Matrix assembling the matrix only called the application operator of the given bilinear form like shown in algorithm 4.4.

Algorithm 4.5 Sequence of operations in contained: is the edge e contained in e' or vice-versa?

```
Called on edge $e$ with edge $e'$ as argument.
  • If $e = e'$, return 1.
  • If level of $e <$ level of $e'$:
      – Coarsen edge $e'$ (level of $e'$ − level of $e$) times, if possible. Otherwise, return 0.
      – If $e = e'$, return 2.
  • If level of $e' <$ level of $e$:
      – Coarsen edge $e$ (level of $e$ − level of $e'$) times, if possible. Otherwise, return 0.
      – If $e = e'$, return 3.
  • Return 0.
```

There already existed a constructor which computed all the necessary element matrices but it did not take into account that the resulting global stiffness matrix was not symmetric.

Solutions

The problems shown above were not equally difficult to solve:

1. To solve this problem, I wrote a new method contained in the class dg_Edge derived from geo_Edge which detects if an edge is contained in another one or vice-versa. The method called on edge e needs as argument an edge e'. The method returns:

   0 if $e \cap e' = \emptyset$.

   1 if $e = e'$.

   2 if $e \supset e'$.

   3 if $e \subset e'$.

   The detection of $e = e'$ is easy as each edge has a unique key. Therefore, $e = e' \Leftrightarrow$ the key of e and e' are equal. To detect either $e \supset e'$ or $e \subset e'$, I introduced a level for each edge. This level indicates on which level of the topology the edges lives. If an edge is a child of an other edge (it was created in a refinement process of an *edge*), then there is a reference to the father of the edge. Otherwise, there is no father.

   With these means, on can implement algorithm 4.5.

   The reference to the father and the level counter make the memory footprint of the topology somewhat larger. There would be a possibility to get the same result without a reference to the father. One could refine the edge with the lower level

Algorithm 4.6 Sequence of operations to assemble a non-symmetric stiffness matrix.

```
• Loop over all elements $K$ of the space:
    – Get the T matrix $T_K$ of $K$.
    – Loop over all elements $K'$ of the space:
        ▷ Compute $A_{KK'}$ by using the application operator of the bilinear form.
        ▷ Get the T matrix $T_{K'}$ of $K'$.
        ▷ Transform the local degrees of freedom into global ones by $\left((A_{KK'}T_{K'})^\top \cdot T_K\right)^\top = T_K^\top A_{KK'}T_{K'}$.
        ▷ Add $\left(T_K^\top A_{KK'}T_{K'}\right)^\top$ entry by entry into the global stiffness matrix (which will be the transposed global stiffness matrix).
```

as many times until it has the same level as the other edge. But this idea has two drawbacks:

- Refining an edge creates two new edges which have to be checked, i. e. a lot of cases arise.
- Refining an edge creates two new edges in the topology, if they are not already present. During the computation of the stiffness matrices, all edges are compared with each other. For a geometric mesh, this means a lot of useless edges would be created in the topology and would not be removed afterwards.

2. The non-symmetric stiffness matrices should be available in transposed form as (1.6) suggests. Algorithm 4.6 does exactly that.

4.3 Other Extensions

4.3.1 Boundary Conditions

During my investigation of the present code of Concepts, I did not find any means to implement variable boundary conditions. Hoch and Rüegg implemented mixed and variable boundary conditions in [15] but this implementation did not suit my needs. Therefore, I did my own implementation of boundary conditions by creating a class dg_BoundaryCond, derived from geo_Attribute (c. f. figure 4.4).

Until now, boundary conditions were only a type, coded as an integer, i. e. the class geo_Attribute represents essentially only a number. Hence, it was possible to include a copy of the boundary condition object in each topological object needing boundary conditions. As I needed at least a function for each boundary condition type, these object would have been significantly too large to be included in every topological object.

That's why I changed the whole idea of boundary conditions in the topology. Now, there is only a reference to a boundary condition stored in each topological object which needs boundary conditions. The boundary condition object itself is only needed once for each boundary condition and is created at the beginning of the mesh creation and destroyed at the end of the mesh removal (c. f. figure 3.16).

The class dg_BoundaryCond takes a type and a function as constructor arguments and has methods to query the type and to compute the value of the function at a certain physical point.

4.3.2 Graph of a Solution

Developing new bilinear and linear forms for Concepts is rather direct and straight forward from the algorithmic point of view. But if you have to code these new bilinear and linear forms, there are many possibilities to make mistakes. Most of the new code I had to write has to do with integration over edges.

When I started my diploma thesis, the only available output was the whole stiffness matrix or the L^2 error at the end of the computation. Whereas the former provides too much, the latter does not provide enough information. On the other hand, a graphical interpretation of the solution vector would be the ideal mean to discover mistakes in the integration over an edge as they show up as large jumps although the solution is smooth and well behaved.

To create such a graphical interpretation of the solution vector there were different things to do:

1. As the elements have the information about the shape functions and can evaluate the solution vector, the solution vector has to be stored in the elements.

   I wrote a new method for the space class dg_HP2d: storeSolution which calls storeSolution on each element. The method storeSolution of an element takes the necessary coefficients out of the solution vector (using the information in the T matrices) and stores them locally.

2. Evaluate the solution in a mesh of points and create a picture of the data.

   I wrote a new method for the element class dg_HP2d001: solutionInPoint which computes the value of the solution in a given point. The method drawGraph in the space class calls this method for the points of a mesh and stores the data in a file which can be read by Gnuplot [19].

The methods as described above can also be used to draw a picture of the error on the domain Ω if the exact solution is known. The coefficients of the exact solution are computed by a L^2 projection and a simple subtraction gives the coefficients of the error.

4.3.3 Sort of Adaptivity

With the linear polynomial degree distribution vector as described in proposition 2.13, most graphs of the error show that the error was not distributed in such a way that the

Algorithm 4.7 Sequence of operations for a geometric mesh in an adaptive algorithm.

```
• Compute the exact solution and the DGFE-solution.
• Store the solution and the weights to compute the error from the solution vector in the elements.
• Loop over all elements and get the maximal element error $e_{\max}$.
• Loop over all elements $K \in \mathcal{T}$:
    – If the element error $e_K > \eta \cdot e_{\max}$ then:
        ▷ If the origin is in the element $K$, then refine it (i. e. subdivide it). Otherwise: raise
          the polynomial degree by 1.
• Start again with the computation on top.
```

linear degree vector is the ideal way of refinement. This raised the idea of an adaptive algorithm.

To make this possible I wrote another set of methods to store the weights to compute the L^2 error. With this, it is possible to compute the element error by a simple scalar product between the coefficients of the solution vector and the weights. This easy way of computing the local error is only possible, if the exact solution is known.

Algorithm 4.7 shows an *hp*-refinement algorithm to create a geometric mesh towards the origin (where the singularity of the exact solution lies).

4.3.4 Debugging Techniques

Debugging a large library like Concepts is sometimes painful. I developed some techniques which I describe in this section.

Using a Debugger

I chose DDD—Data Display Debugger [20] as my favorite debugger because of its excellent way to graphically display complicated data structures.

Using a debugger is not always the best choice. Above all, if you want to have a quick overview what happens at a certain point in the code in a large loop, selective output on screen is much better suited.

Output on Screen

To handle output on screen efficiently, I introduced two compiler macros in the top level include file `debug.h`: `DP` and `DPL`. The first one stands for “debug print” and the second stands for “debug print line”. Pristinely, these are not my idea but are inspired by Rolf Negri, an assistant at the Institute for Operations Research at the ETH.

These macros both have three arguments: an integer to turn the output on and off, a text and an arbitrary variable. Defining the integer in a package level include file (e. g. `dgDebug.h` for the DGFEM package) gives an as fine grained control over the debugging messages as one wishes to have.

For a production build of the library, the compiler define `DEBUG` can be omitted and all macros `DP` and `DPL` are ignored.

Assertions

A third mean to debug a code is to use assertions. If an assertion fails, the execution of the program is immediately terminated with a verbose error message describing the file and the line in the file where the assertion failed. These assertions are only active in a debugging build of the library and don't slow the execution in a production build.

Assertions should be used to check the prerequisites of a method, e. g. the range of parameters.

4.4 Wish List for Extensions

The first paragraph lists some possible technical extensions to Concepts which do not really extend the functionality but rather make it easier to add new components. The second paragraph lists some wishes which arose during my diploma thesis.

4.4.1 Technical Extensions

Lage, the main author of Concepts, who is still actively developing and maintaining it, plans to use more C++ features which were not stable when he started the development of Concepts. Namely, these are exceptions (error handling), name spaces (to remove the ugly `geo_`, `hp_` etc. prefixes) and templates (i. e. parametrized classes).

4.4.2 Algorithmic and Functional Extensions

The extensions to Concepts I described in this chapter are by far not complete:

- The bilinear and linear forms described in section 4.1 can only cope with problems with constant coefficients. Therefore, a next step could be to implement *non-constant coefficients* for the diffusion, advection and the absolute term.

  For the diffusion term, even a symmetric matrix of coefficients would be desirable.

- The theoretical proof of the exponential convergence in chapter 2 uses *strongly enforced Dirichlet boundary conditions* whereas the variational formulation in section 1.3.1 used as the basis for the implementation of DGFEM uses weakly enforced

Dirichlet boundary condition. The best solution would be, if the user could chose between weakly and strongly enforced boundary conditions.

Because of this difference, the results presented in the next chapter are not as significant as they could be if the boundary conditions were implemented differently. But implementing strongly enforced boundary conditions takes more time than there is for a diploma thesis.

- The graphics of the mesh (which are already part of the class hp_HP2d) are not very good looking. The graph of the solution described in this chapter does not show the mesh at all. *More sophisticated graphics* should include the mesh and the solution in one image.

  Another enhancement which also belongs to the domain of user interface is a *mesh generator*. Currently, the meshes are coded by hand in a specialized class, for instance dg_LShape in the appendix on page 78 and following.

- The element class dg_HP2d001 is derived from the FE-class hp_HP2d001. Therefore, it uses the same shape functions as the FE-code. This is not really a sensible choice, since the terms with the normal derivative in the bilinear form (1.16) make all shape functions exterior shape functions, i. e. they all interact with the neighbouring elements. Choosing *doubly integrated Legendre polynomials as shape functions* results in only 16 exterior shape functions, the rest are internal shape functions. The degrees of freedom associated with internal shape functions can be eliminated with static condensation. On the other hand, this choice implies that the polynomial degree is at least three.

  The necessary changes to implement these shape functions would affect the rebuild process in the space class (where the degrees of freedoms are counted) and the element class (where the shape functions and their derivatives are evaluated).

  See [8] for a short summary on shape functions based on the doubly integrated Legendre polynomials.

- Implementing better suited shape functions would have the benefit that *static condensation* was possible. This idea is completely missing in Concepts right now, also in the FEM code.

- The shape functions on the quadrilaterals have a tensor product Ansatz:

$$N_i(\xi) = N_i(\xi_1, \xi_2) = M_k(\xi_1) \cdot M_l(\xi_2),$$

  where $\{M_K\}_k$ are one dimensional shape functions on $(0, 1)$. This property can be exploited during the element computations with so called sum factorization [6]. The benefit is significantly reduced computation time for the element matrices for high polynomial degrees.

In Concepts, the shape functions in `hp_HP2d001` are currently not available as factors M_k and M_l but only as product N_i. This has to be changed if sum factorization should be implemented.

- If only few elements are changed, the new solution should be not so far from the old solution—at least in the elements which have not changed. Therefore, saving the old solution and taking it as a starting value for an iterative solve for the new solution should save some computation time. This is would be a first step towards a *multi grid algorithm*.

  For a true multi grid algorithm, the new solution on changed elements has to be approximated by an interpolation of the old solution. Then the solver itself works iteratively on different meshes to approximate the solution of the linear system.

5 Numerical Results

In this chapter, the numerical results which should provide the evidence of the theoretical results in chapter 2 are presented. The model problem is introduced in the first section. The results on the unit square and on a L shaped domain are given in the consecutive sections. In the last section, the conclusions are drawn from the computations: does the implemented variational formulation from section 1.3.1 provide exponential convergence? Also in the last section of this chapter: the impact of the stabilisation on the error and the condition number of the stiffness matrix.

All computations in the second and third section have been done without stabilisation. The error of the numerical solution is shown in two different norms:

- The error in the energy norm: $|u - u_{DG}|_{DG}^2 = B(u - u_{DG}, u - u_{DG})$. Observe that this norm is not the same for the stabilised version and the version without stabilisation: the bilinear form B is taken from (1.26) with and without the stabilisation term respectively.

- The error in the L^2-norm: $\|u - u_{DG}\|_0^2 = \int_\Omega |u - u_{DG}|^2 \, dx$. This norm does not depend on the stabilisation.

5.1 Model Problem

In order be able to compute the error without complicated error estimation, I chose the exact solution and calculated the according right hand side. The exact solution should have some sort of singularity at one point. There are many different possibilities, e. g. $u = \sqrt{r}$, where $r^2 = (x^1)^2 + (x^2)^2$, i. e. r is the distance from the origin.

The domains of interest are the unit square and an L shaped domain (see figures 5.6 and 5.7 on page 52 respectively). The exact solutions are shown in figure 5.1 and figure 5.2 for the unit square and the L shaped domain respectively.

Inserting $a = 1$, $\underline{b} = (0, 0)$, $c = 1$ and $u = \sqrt{r}$ into (1.13) gives:

$$\frac{3(x^1)^2}{4\sqrt{r}^7} + \frac{3(x^2)^2}{4\sqrt{r}^7} - \frac{1}{\sqrt{r}^3} + \sqrt{r} = f. \tag{5.1}$$

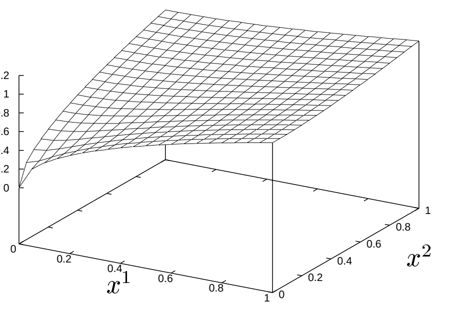


Figure 5.1: The exact solution on the unit square $(0,1)^2$.

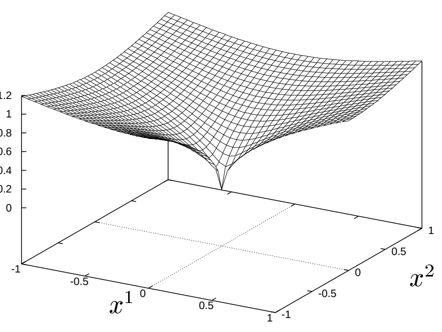


Figure 5.2: The exact solution on the L shaped domain.

The boundary conditions on the unit square are of Dirichlet type on the coordinate axes and of Neumann type otherwise. Hence, the Dirichlet boundary conditions are

$$u|_{x^1=0} = \sqrt{x^2} \text{ and } u|_{x^2=0} = \sqrt{x^1}$$

and the Neumann boundary conditions are

$$\left.\frac{\partial u}{\partial \underline{n}}\right|_{x^1=\pm 1} = \frac{1}{2(1+(x^2)^2)^{3/4}} = \frac{x^1}{2\sqrt{r}^3} \text{ and } \left.\frac{\partial u}{\partial \underline{n}}\right|_{x^2=\pm 1} = \frac{1}{2(1+(x^1)^2)^{3/4}} = \frac{x^2}{2\sqrt{r}^3}.$$

5.2 Results on the Unit Square

The results on the unit square $(0,1)^2$ for $u = \sqrt{r}$ are presented in the following.

The meshes in pictures like figure 5.3 or 5.4 show the element boundaries and each element is coloured according to its polynomial degree: the darker an element the higher its polynomial degree.

h-version DGFEM

In the h-version DGFEM, only the mesh is refined and the polynomial degree is kept constant.

There are basically two methods to perform the mesh refinement: uniformly on the whole domain of interest or adaptively on the elements with the highest error. The uniform refinement is easy to implement, because all elements can be treated in the same way. The adaptive refinement is not much harder because the exact solution is known. Algorithm 4.7 shows an adaptive algorithm to create a geometric mesh towards the origin in the unit square. This algorithm can also be used to perform an adaptive h-refinement if every element with a large error is subdivided.

Figure 5.5 shows the results of some calculations on the unit square. Figure 5.3 shows the mesh after five adaptive h-refinement steps with $p = 1$.

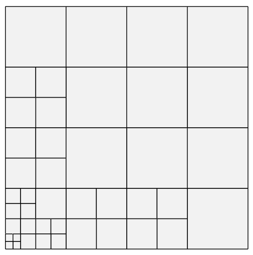


Figure 5.3: The mesh on the unit square after five adaptive h-refinement steps.

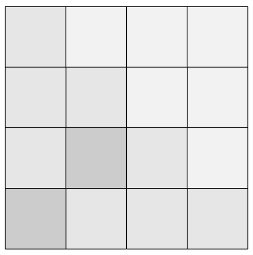


Figure 5.4: The mesh on the unit square after five adaptive p-refinement steps on an initial mesh with 16 elements.

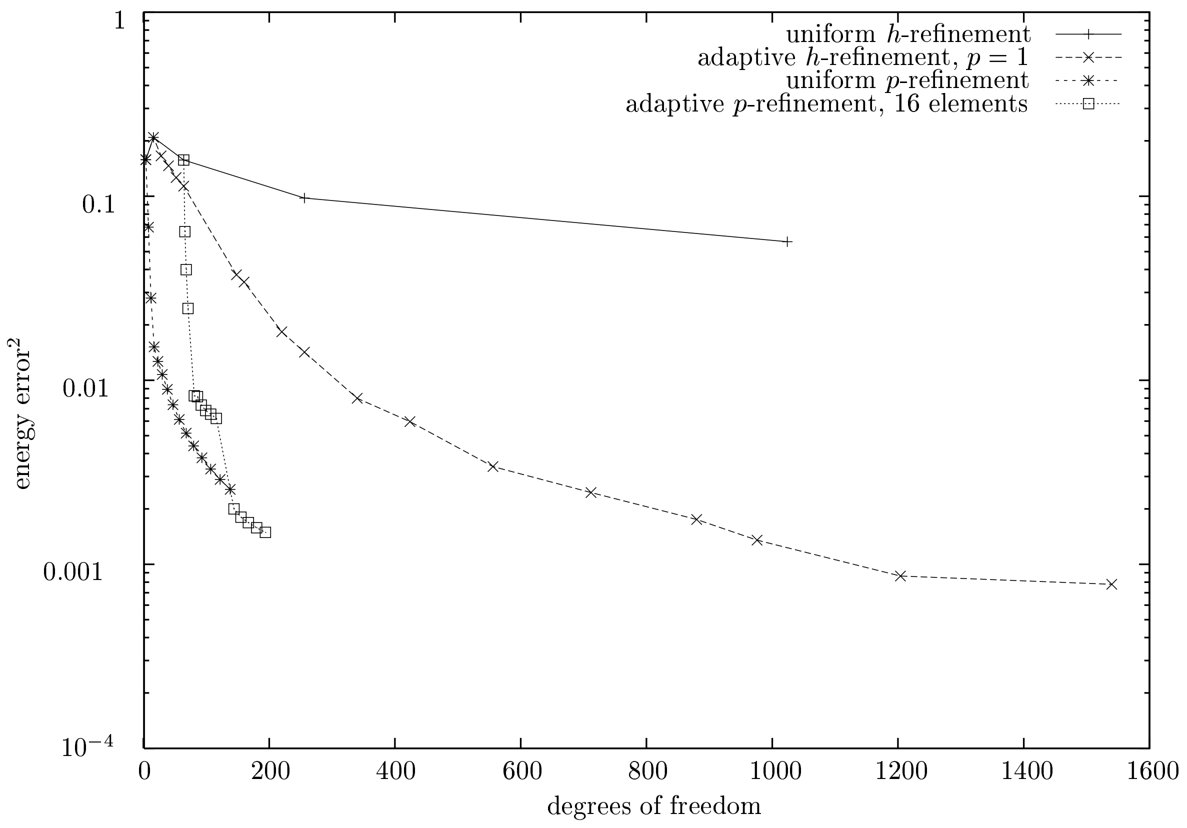


Figure 5.5: The error $|u - u_{DG}|_{DG}^2$ of the h- and the p-version DGFEM on the unit square.

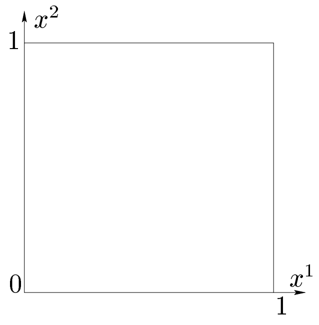


Figure 5.6: The unit square $(0,1)^2$.

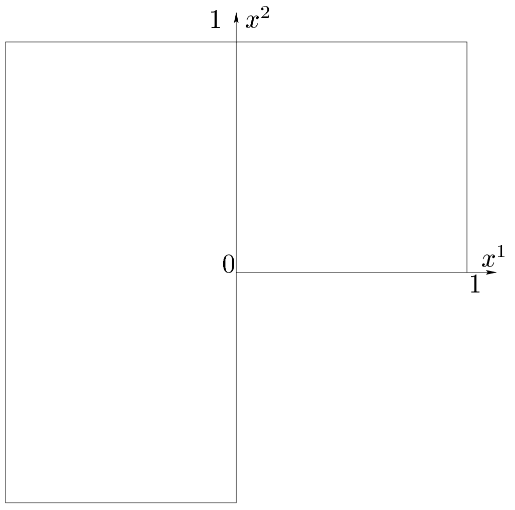


Figure 5.7: An L shaped domain.

p-version DGFEM

In the p-version DGFEM, only the polynomial degree is raised and the mesh is left unrefined.

As in the h-version DGFEM, there are two methods to refine the mesh: uniformly on all elements or adaptively depending on the element error. The uniform refinement was performed on a one-element-mesh, the adaptive refinements on a mesh with 16 elements. Figure 5.4 shows the mesh with 16 elements after four refinement steps. Figure 5.5 shows the results of the p-version DGFEM. The curves don't go further because the maximal polynomial degree of 15 was reached during the computations.

hp-version DGFEM

In the hp-version DGFEM, the polynomial degree is raised and the mesh is refined at the same time (but not necessarily on the same element).

Again, there are two methods to refine the mesh: adaptively (c. f. algorithm 4.7) or with a linear degree vector (c. f. proposition 2.13).

Figure 5.8 shows the results of the hp-version DGFEM with the linear degree vector. The curve for $\mu = 2$ stops early because the maximal polynomial degree of 15 was reached, whereas the curve for $\mu = 1/2$ stops early because the error was growing instead of falling for more than 800 degrees of freedom.

Figure 5.9 shows the results of the hp-version DGFEM with the adaptive algorithm 4.7. Whereas for the linear degree vector above, I only refined towards the origin, I made some more experiments with the adaptive algorithm. I refined towards one and three corners (see figure 5.10 for an idea of the meshes) and modified the parameter

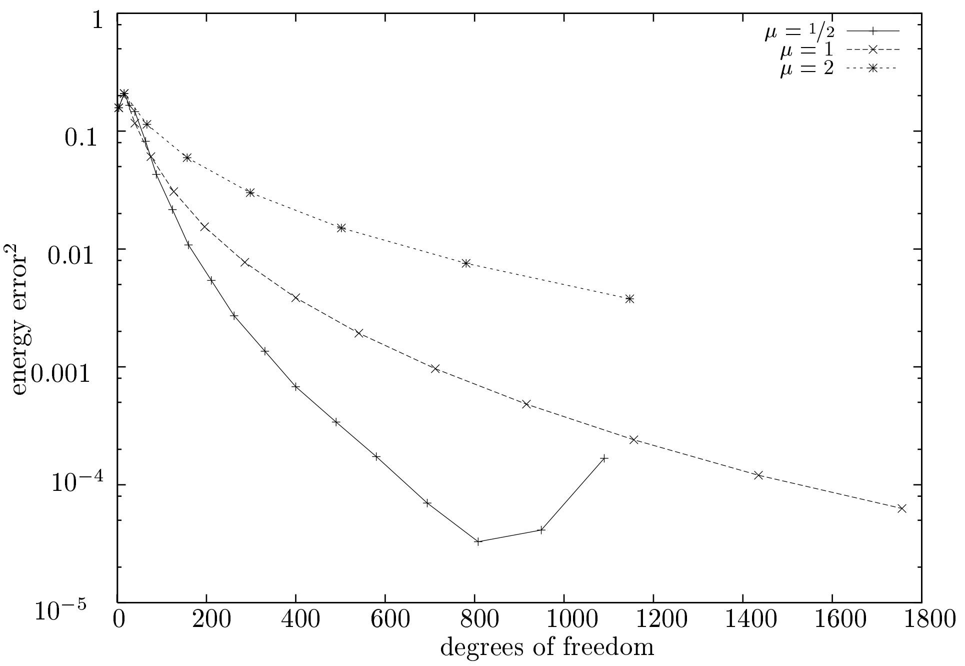


Figure 5.8: The error of the *hp*-version DGFEM with the linear degree vector on the unit square.

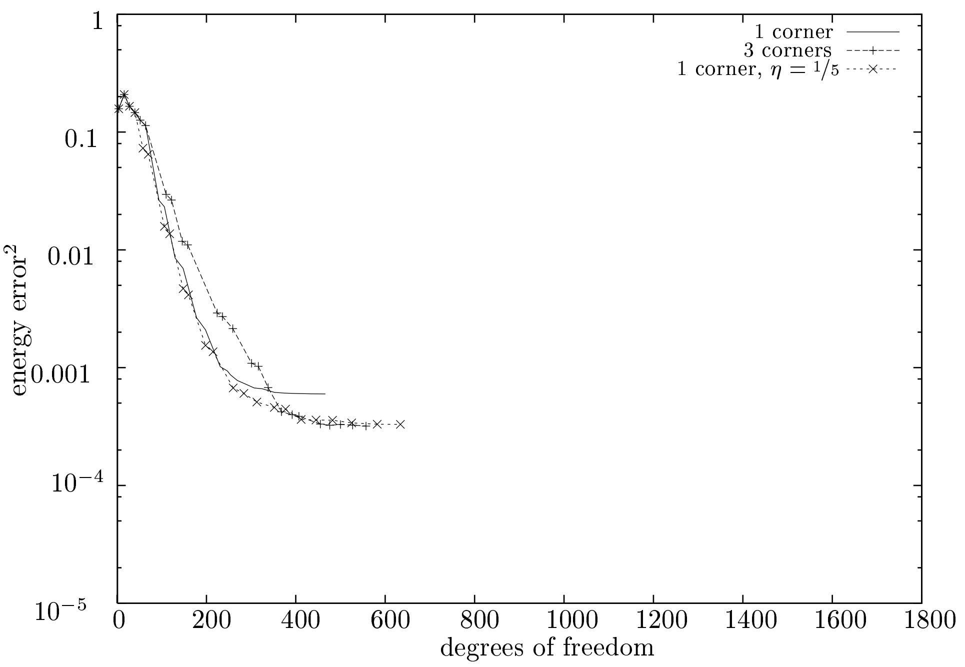
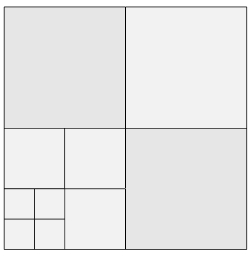


Figure 5.9: The error of the *hp*-version DGFEM with the adaptive algorithm on the unit square. Normally, $\eta = 1/2$ unless stated otherwise.


(a) Refinement towards one corner in $(0,1)^2$.

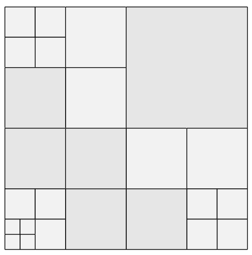

(b) Refinement towards three corners in $(0,1)^2$: towards all corners but $(1,1)$.

Figure 5.10: Refinement towards corners of the unit square.

η which controls how large the error of an element is allowed to be in order that the element is not changed (i. e. is not refined and the polynomial degree is not changed), see algorithm 4.7 for details. All curves reached the maximal polynomial degree of 15.

For the curves in the figures above, if not already stated otherwise, only computation time and memory demand were a problem, i. e. the matrices grew too large to be solved in acceptable time.

Comparison

The comparisons in figure 5.11 and 5.12 show that the *hp*-version is far superior compared to the *h*- or the *p*-version DGFEM. It does not matter much if with an adaptive algorithm or with a linear degree vector, tough.

5.3 Results on an L Shaped Domain

The results in this section are not as detailed as those in the previous section because we already have an idea which parameters give good convergence.

The results of the *hp*-version DGFEM with the linear degree vector and the adaptive algorithm are shown in figure 5.13. To get an idea of the used meshes during the computations, see figure 5.14.

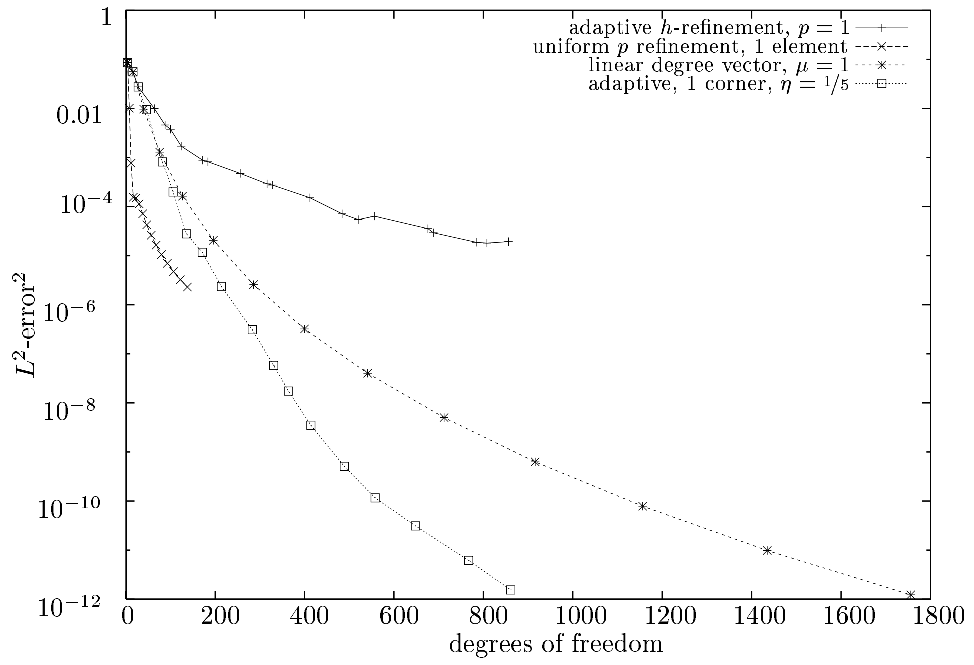


Figure 5.11: The error of the h-, p- and hp-version DGFEM on the unit square in the L^2-norm.

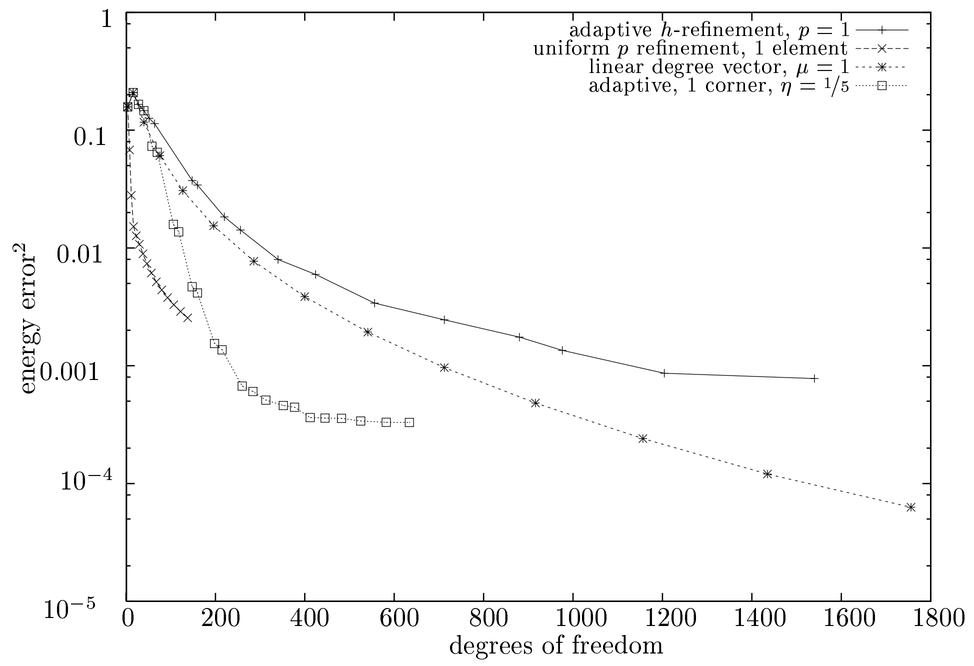


Figure 5.12: The error of the h-, p- and hp-version DGFEM on the unit square in the energy norm.

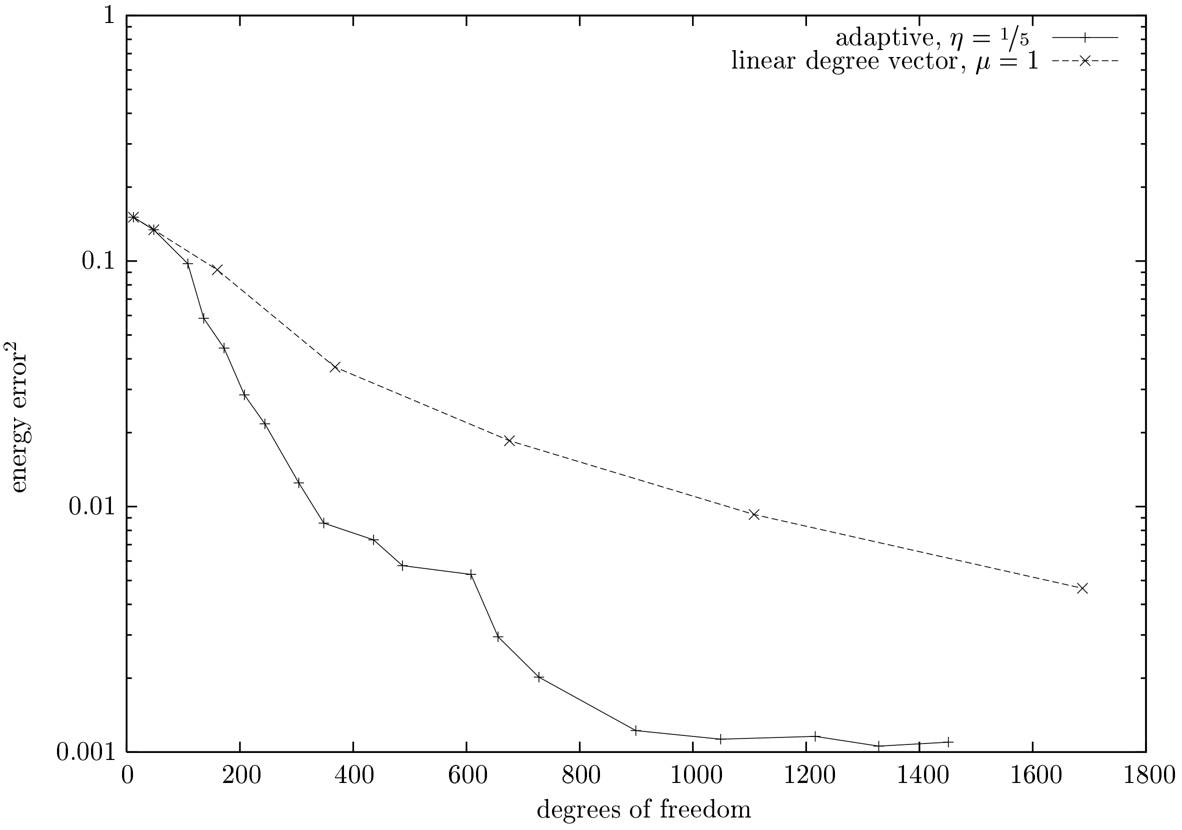
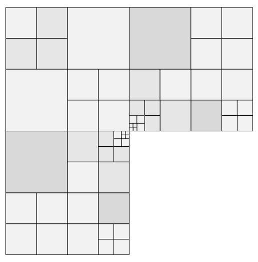
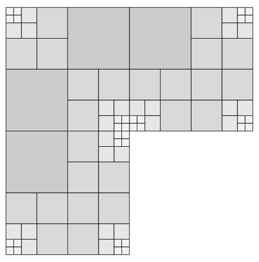


Figure 5.13: The error of the *hp*-version DGFEM on the L shaped domain.



(a) Refinement of the L shaped domain with the adaptive algorithm.



(b) Refinement of the L shaped domain with the linear degree vector.

Figure 5.14: Refinement of the L shaped domain.

Problem	$2b$	C^2	Fit
unit square, linear degree vector, $\mu = 1$	0.87072	2.2937	yes
unit square, adaptive, 1 corner, $\eta = 1/5$	1.17860	8.2348	no
L shape, linear degree vector, $\mu = 1$	0.46106	1.1255	yes
L shape, adaptive, $\eta = 1/5$	0.78555	4.1134	no

Table 5.1: Parameters of the interpolating curves in figure 5.15.

5.4 Conclusions

5.4.1 Exponential Convergence

The results in chapter 2 predict exponential convergence for the linear degree vector, i. e.

$$|u_{\text{ex}} - u_{DG}|_{DG} \leq Ce^{-b\sqrt[3]{N}}.$$

All convergence plots in this chapter are semi-logarithmic because one can see exponential convergence of the form error $\leq Ce^{-bN}$ quite clearly as a straight line, since

$$\begin{aligned} \text{error} &\leq Ce^{-bN} \\ \ln(\text{error}) &\leq \ln C - bN, \end{aligned}$$

where N stands for the degrees of freedom. Unfortunately, the predicted error bound has got this $\sqrt[3]{N}$ which results in

$$\begin{aligned} |u_{\text{ex}} - u_{DG}|_{DG} &\leq Ce^{-b\sqrt[3]{N}} \\ \ln(|u_{\text{ex}} - u_{DG}|_{DG}) &\leq \ln C - b\sqrt[3]{N} \end{aligned}$$

which cannot be identified easily in such a plot. Plotting the error against $\sqrt[3]{N}$ makes this easier: the curve should be a straight line there.

To make the identification even easier, I have calculated the parameters b and C from two points in the plots and included the resulting graphs in figure 5.15 for both the L shaped domain and the unit square. All plots are against the square of the error, therefore

$$\ln(|u_{\text{ex}} - u_{DG}|^2_{DG}) \leq \ln C^2 - 2b\sqrt[3]{N}{}^b\!/_2\sqrt[3]{N}$$

The results of the interpolation are summed up in table 5.1. Although the meshes in the proof in chapter 2 and in Concepts are slightly different, the method using the linear degree vector yields exponential convergence like predicted.

The adaptive method (c. f. algorithm 4.7) does not yield exponential convergence but according to figure 5.15, they reach the same absolute error with much less degrees of freedom up to a certain bound. May be, there is more sophisticated algorithm which keeps the convergence rate from flattening for growing number of degrees of freedom.

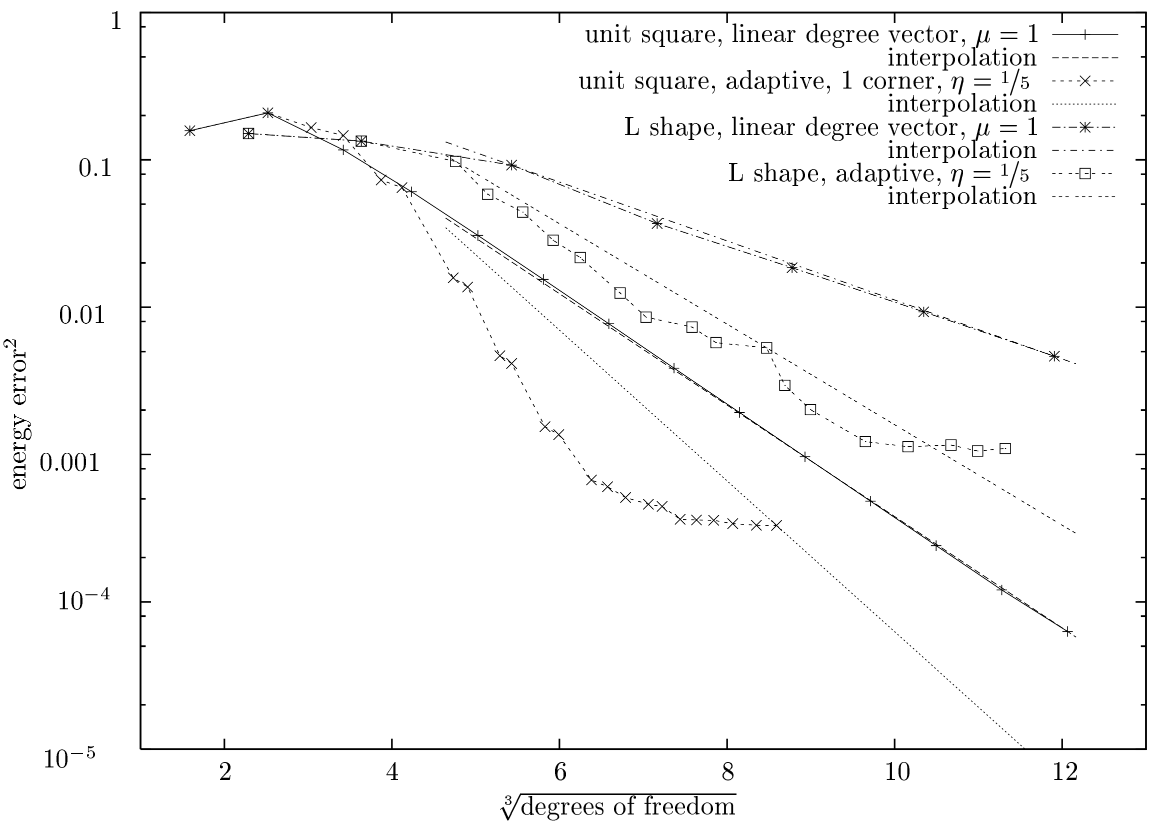


Figure 5.15: Conclusion on the unit square and on the L shaped domain. The interpolation curve is drawn without points interpolating the curve just above in the legend of the plot.

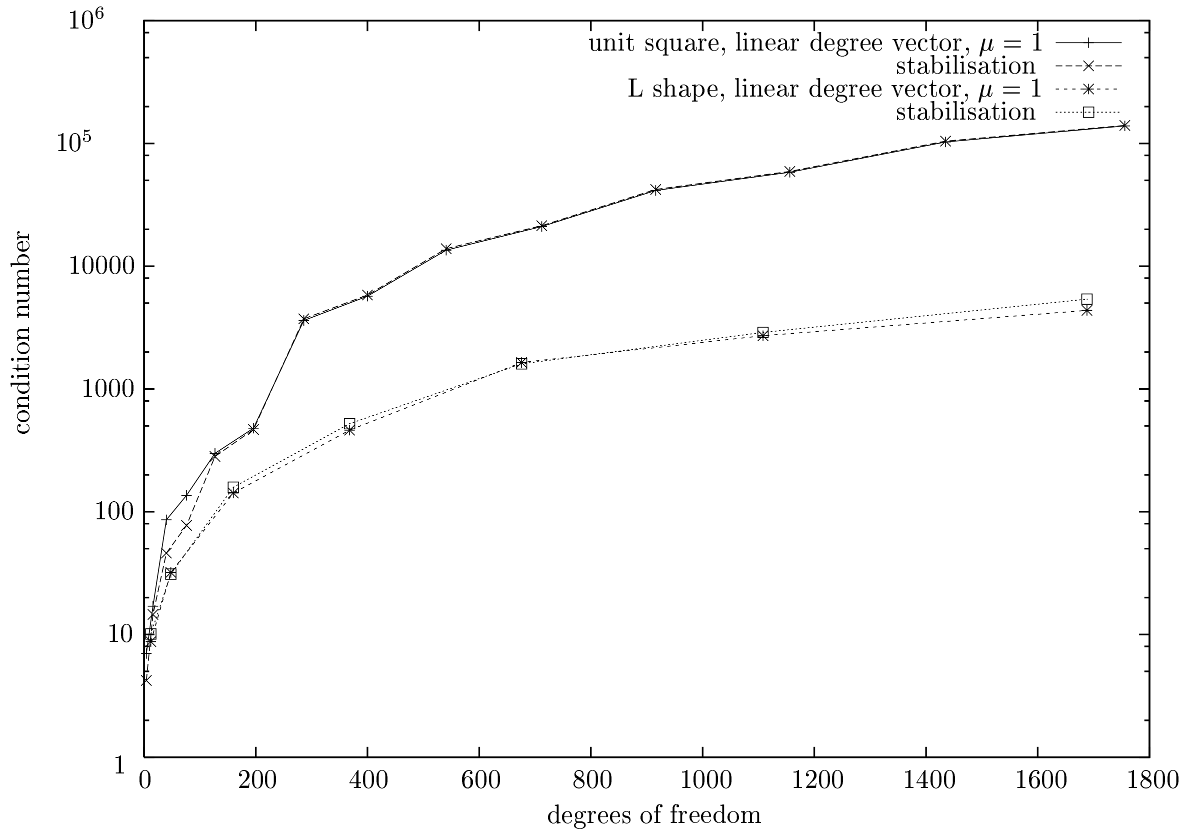


Figure 5.16: The impact of stabilisation on the condition number of the stiffness matrix. The stabilised version of a curve is the one just below in the legend of the plot.

5.4.2 Stabilisation

Up to here, all computations were done without the stabilisation from (1.23) or (2.6). Nevertheless, figure 5.15 shows exponential convergence.

The condition number of the stabilised stiffness matrices does not improve compared to the non-stabilised matrices. Figure 5.16 shows the impact of stabilisation on the condition number of the stiffness matrices.

The figures 5.17 (for the L^2-error) and 5.18 (for the energy error) show the impact of stabilisation on the error. Both on the unit square and on the L shaped domain and both for the L^2-error and the energy error, the stabilised method is somewhat better. The improvement is not in the convergence rate b but only in the constant C, tough. This conclusion can be drawn without computing the parameters b and C: the respective lines in the plots are parallel.

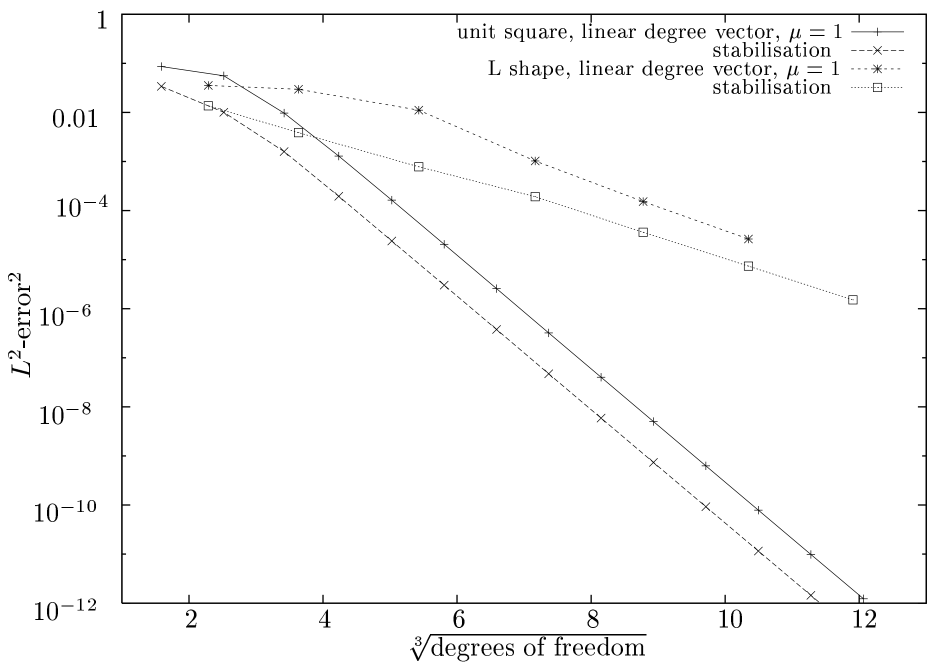


Figure 5.17: The impact of stabilisation on the L^2-error. The stabilised version of a curve is the one just below in the legend of the plot.

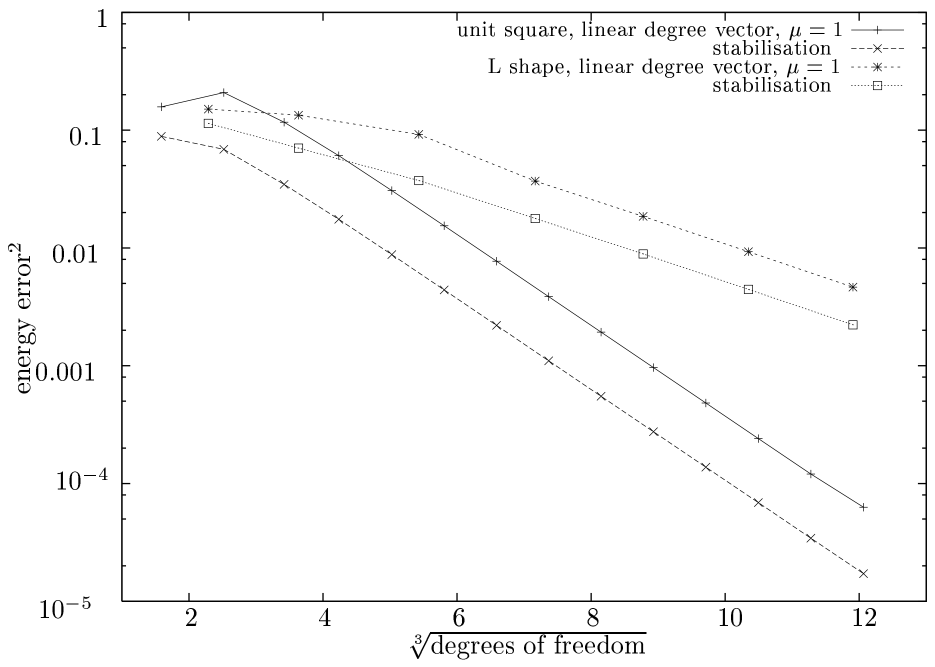


Figure 5.18: The impact of stabilisation on the energy error. The stabilised version of a curve is the one just below in the legend of the plot.

Closing Remarks

The *theoretical part* in chapter 2 proofs the exponential convergence of the *hp*-version of the DGFEM with a linear degree vector [18], i. e. the vector of polynomial degrees on the mesh is given by

$$\underline{p} = \left\{p_{ij} = p_i, i = 0, \ldots, n, j = 1, 2, p_i = \max\left\{1, \lfloor \mu i \rfloor\right\}\right\},$$

where p_{ij} is the polynomial degree of the j^{th} element on level i of the mesh, see figure 2.1 on page 13 and proposition 2.13 on page 12. This defines the standard mesh on the reference element $\hat{\Omega}$ (the unit square $(0,1)^2$). On a polygon, near each vertex of the polygon, the mesh is constructed in the same way as on the reference element: see definition 2.14 on page 13. With such a mesh, the convergence is exponential:

$$\left|u_{\text{ex}} - u_{DG}\right|_{DG} \leq Ce^{-b\sqrt[3]{N}},$$

where N is the number of degrees of freedom, u_{ex} the exact solution of the problem and u_{DG} the DGFEM-approximation. The problem in this case is of the form

$$-\nabla(a\nabla u) + cu = f \text{ in } \Omega, u = 0 \text{ on } \Gamma_D \text{ and } (a\nabla u) \cdot \underline{n} = g_N \text{ on } \Gamma_N.$$

In the *practical part* in chapter 5, I found the same convergence rate for the model problem $u_{\text{ex}} = \sqrt{r}$ on the unit square $(0,1)^2$ and on an L shaped domain (c. f. figure 5.7 on page 52), no matter if stabilised or not. These results are presented in section 5.4. The equations which I implemented are not exactly the same as in the theoretical part, though. These differences are summarized in the following table:

	Theory	Practice
Mesh	figure 2.1, page 13	figure 5.10, page 54
Dirichlet boundary conditions	strongly enforced	weakly enforced

Note of Thanks

I would like to thank the following people for helping me with my diploma thesis: Ana-Maria Matache (she helped me to understand the software Concepts), Thomas Wihler (he checked some of my computations and he and Prof. Schwab are writing the paper [18] from which I took the theoretical part) and Prof. Schwab (despite he had to instruct me, he had a lot of confidence in me).

A Documented Source Code of the Extensions to Concepts

A.1 Integration over an Edge

As described in section 4.2.2, to detect common edges of two elements, I wrote the class dg_Edge. The header and the implementation file for dg_Edge are printed below.

In addition to the class dg_Edge, the class dg_Quad was needed for two reasons:

- When refining a quadrilateral, all new edges have to be of the type dg_Edge.
- The implementation of the linear polynomial degree distribution vector described in chapter 2 needs the level of each element. This level information is added in dg_Quad.

dgTopology.h

```
/* -*- c++ -*-
 * Topology for DG FEM
 * connectivity informations
 */

#ifndef dgTopology_h
#define dgTopology_h

#include <iostream.h>
#include <typeinfo>

#include "../concepts.h"
#include "../geometry/geoTopology.h"

// ************************************************************* dg_Edge **

/** An edge in the topology of DGFEM. Additional level and father
    information to detect elements which share common edges. */
class dg_Edge : public geo_Edge {

public:
  /** Constructor. Creates an edge out of two vertices. The constructor of
      geo_Edge is called to create the edge. Then, the level and father
      information is stored. */
  dg_Edge(geo_Vertex& vtx0, geo_Vertex& vtx1,
          geo_Attribute* attrib = 0,
          const dg_Edge* father = 0, const uint level = 0);
  virtual ~dg_Edge() {}

  /** Returns a child. If no child exists, two new children are
      created (with the same attributes as this one). The new edges all
      have a level which is increased by one. Their father pointer is
      also initialized with the right value. */
  virtual geo_Edge* child(uint i);

  /// Returns information in an output stream
  virtual ostream& info(ostream& os) const;

  /** Returns a pointer to the father. */
  inline const dg_Edge* father() const {
    return father_;
  }

  /** Returns the relation of the two edges, ie. if one is contained in
      the other.

      This method is new and not yet
      fully stable, ie. it can move or change its name. Better suited
      would be one level up in the hierarchy (ie. geo_Connector1).

      @return \begin{itemize}
      \item 0: Neither of the edges is contained in the other
      \item 1: The edges are equal
      \item 2: The other edge is contained in this edge
      \item 3: This edge is contained in the other edge
      \end{itemize}
      @param other The other edge */
  const uint contained(const dg_Edge& other) const;

private:
  /** Pointer to the father edge. The father information is stored
      on creation and not changed afterwards. */
  const dg_Edge* father_;

  /** The level of the edge. The level information is stored
      on creation and not changed afterwards. */
  const uint level_;

  geo_Edge*& getLnk_() {
    return lnk_;
  }
```

```
};

// ************************************************************** dg_Quad **

/** A quadrilateral in the topology. Additional level information to
    make meshes with linear polynomial degree destribution vector
    possible. */
class dg_Quad : public geo_Quad {

public:
  /** Constructor. Creates a quadrilateral out of four edges.
      The constructor of geo_Quad is called to create the quadrilateral,
      then the level information is stored. */
  dg_Quad(geo_Edge& edg0, geo_Edge& edg1, geo_Edge& edg2, geo_Edge& edg3,
          geo_Attribute* attrib = 0, const uint level = 0);
  virtual ~dg_Quad() {}

  /** Returns a child. If no childs exist, four new children are created
      (with the same attributes as this one). All edges of the quadrilateral
      are refined and four new edges introduced. The new quadrilaterals
      all have a level which is increased by one. */
  virtual geo_Quad* child(uint i);

  /** Returns the level of the quadrilateral. */
  inline const uint level() const {
    return level_;
  }

private:
  /** The level of the quad. The top level in the inital mesh is 0,
      the next finer level is 1 etc. The level information is stored
      on creation and not changed afterwards. */
  const uint level_;

  geo_Quad*& getLnk_() {
    return lnk_;
  }
};

#endif // dgTopology_h
```

dgTopology.cc

```
/* Topology for DG FEM
 * connectivity information
 */

#include "dgTopology.h"
#include "dgDebug.h"
```

```
// *********************************************************** dg_Edge **

dg_Edge::dg_Edge(geo_Vertex& vtx0, geo_Vertex& vtx1,
                 geo_Attribute* attrib = 0,
                 const dg_Edge* father = 0, const uint level = 0) :
  geo_Edge::geo_Edge(vtx0, vtx1, attrib), father_(father), level_(level) {

  DPL(dgEdgeConstr_D, "[" << __FILE__ << ", line " << __LINE__
      << "] dg_Edge::dg_Edge -- new", *this);
}

geo_Edge* dg_Edge::child(uint i) {
  if (i > 1) return NULL;

  if (chld_ == NULL) {
    DPL(dgEdgeChild_D, "[" << __FILE__ << ", line " << __LINE__
        << "] dg_Edge::child -- refining", *this);

    geo_Vertex* vtx = new geo_Vertex(attrib_);
    chld_ = new dg_Edge(*vtx_[0]->child(0), *vtx, attrib_,
                        this, level_+1);
    dg_Edge* chld = (dg_Edge*)chld_;
    chld->getLnk_() = new dg_Edge(*vtx, *vtx_[1]->child(0), attrib_,
                                  this, level_+1);
  }

  if (i == 0)
    return chld_;
  else {
    dg_Edge* chld = dynamic_cast<dg_Edge*>(chld_);
    if (chld)
      return chld->getLnk_();
    else {
                                                  // *** exception ***
      cerr << '[' << __FILE__ << ", line " << __LINE__;
      cerr << "] dg_Edge::child() -- edge not supported\n";
      cerr << " type of child: " << typeid(*chld_).name() << endl;
      exit(1);
    }
  }
}

ostream& dg_Edge::info(ostream& os) const {

  os << "Edge(" << key_ << ", (";
  os << *vtx_[0] << ", " << *vtx_[1] << ", Level=" << level_ << ')';
  return os;
}
```

```
const uint dg_Edge::contained(const dg_Edge& other) const {

  if (*this == other)
    return 1;

  uint i;
  const dg_Edge* me = this;
  const dg_Edge* you = &other;

  DPL(dgEdgeContained_D, "[" << __FILE__ << ", line " << __LINE__
      << "] dg_Edge::contained -- me: " << *me << " at level " << level_
      << ", you: " << *you << " at level", you->level_);

  // compare the levels of the two edges
  if (other.level_ < level_) {
    DPL(dgEdgeContained_D, "[" << __FILE__ << ", line " << __LINE__
        << "] dg_Edge::contained -- coarsening " << *me << " "
        << level_ - other.level_ << " times.", " ");
    for (i = 0; i < (level_ - other.level_); ++i) {
      if (me)
        me = me->father();
      else
        return 0;
    }
    if (me && you && (*me == *you))
      return 3;
  } else
    if (level_ < other.level_) {
      DPL(dgEdgeContained_D, "[" << __FILE__ << ", line " << __LINE__
          << "] dg_Edge::contained -- coarsening " << *you << " "
          << other.level_ - level_ << " times.", " ");
      for (i = 0; i < (other.level_ - level_); ++i) {
        if (you)
          you = you->father();
        else
          return 0;
      }
      if (me && you && (*me == *you))
        return 2;
    }

  return 0;
}

// ************************************************************ dg_Quad **

dg_Quad::dg_Quad(geo_Edge& edg0, geo_Edge& edg1, geo_Edge& edg2,
                 geo_Edge& edg3, geo_Attribute* attrib = 0,
                 const uint level = 0) :
  geo_Quad::geo_Quad(edg0, edg1, edg2, edg3, attrib), level_(level) {
```

```
  DPL(dgQuadConstr_D, "[" << __FILE__ << ", line " << __LINE__
      << "] dg_Quad::dg_Quad --", "new quad");
}

geo_Quad* dg_Quad::child(uint i) {

  dg_Quad* chld;

  if (i > 3) return NULL;

  if (chld_ == NULL) {
    geo_Edge* edgA = edg_[0]->child(rho_[0]);
    geo_Edge* edgB = edg_[0]->child(rho_[0].succ());
    geo_Edge* edgC = edg_[1]->child(rho_[1]);
    geo_Edge* edgD = edg_[1]->child(rho_[1].succ());
    geo_Edge* edgE = edg_[2]->child(rho_[2]);
    geo_Edge* edgF = edg_[2]->child(rho_[2].succ());
    geo_Edge* edgG = edg_[3]->child(rho_[3]);
    geo_Edge* edgH = edg_[3]->child(rho_[3].succ());

    geo_Vertex* vtx4 = edgB->vertex(rho_[0]);
    geo_Vertex* vtx5 = edgD->vertex(rho_[1]);
    geo_Vertex* vtx6 = edgF->vertex(rho_[2]);
    geo_Vertex* vtx7 = edgH->vertex(rho_[3]);

    geo_Vertex* vtx8 = new geo_Vertex(attrib_);

    dg_Edge* edgI = new dg_Edge(*vtx4, *vtx8, attrib_, 0, level_+1);
    dg_Edge* edgJ = new dg_Edge(*vtx5, *vtx8, attrib_, 0, level_+1);
    dg_Edge* edgK = new dg_Edge(*vtx6, *vtx8, attrib_, 0, level_+1);
    dg_Edge* edgL = new dg_Edge(*vtx7, *vtx8, attrib_, 0, level_+1);

    geo_Quad** qd = &chld_;

    *qd = new dg_Quad(*edgA, *edgI, *edgL, *edgH, attrib_, level_+1);
    chld = (dg_Quad*)(*qd);
    qd = &(chld->getLnk_());

    *qd = new dg_Quad(*edgB, *edgC, *edgJ, *edgI, attrib_, level_+1);
    chld = (dg_Quad*)(*qd);
    qd = &(chld->getLnk_());

    *qd = new dg_Quad(*edgJ, *edgD, *edgE, *edgK, attrib_, level_+1);
    chld = (dg_Quad*)(*qd);
    qd = &(chld->getLnk_());

    *qd = new dg_Quad(*edgL, *edgK, *edgF, *edgG, attrib_, level_+1);
  }
  dg_Quad* qd = dynamic_cast<dg_Quad*>(chld_);
  if (! qd) {
                                                    // *** exception ***
    cerr << '[' << __FILE__ << ", line " << __LINE__;
    cerr << "] dg_Quad::child() -- quad not supported\n";
    exit(1);
  }
  while (i--) {
    qd = dynamic_cast<dg_Quad*>(qd->getLnk_());
    if (! qd) {
                                                    // *** exception ***
      cerr << '[' << __FILE__ << ", line " << __LINE__;
      cerr << "] dg_Quad::child() -- quad not supported\n";
      cerr << " type of child: " << typeid(*qd->getLnk_()).name() << endl;
      exit(1);
    }
  }

  return qd;
}
```

Edge Integration Subroutine

The following subroutine shows how the integral

$$\int_e u\big((a\nabla v)\cdot \underline{n}_e\big)\, ds$$

is computed. This subroutine is part of the class dg_BDiffusion. The parameters have the following meaning:

elmX The first element, the shape functions u have their support in elmX.

i Number of the edge in elmX over which has to be integrated.

elmY The second element, the shape functions v have their support in elmY.

j Number of the edge in elmY over which has to be integrated.

em The element matrix.

signa An arbitrary constant which can be given for the integration. Values of 1 (for $e \subset \Gamma_{\text{int}}$) and 2 (for $e \subset \Gamma_D$) occur.

relation The relation of edge i of elmX to edge j of elmY. The following values are possible:

0 if $e \cap e' = \emptyset$, the subroutine is not called if relation is 0.

1 if $e = e'$.

2 if $e \supset e'$.

3 if $e \subset e'$.

To get an idea what happens in the code, see section 3.5, where the integration of $\int_K \nabla u \nabla v \, dx$ is explained, especially the calculation of the derivatives and the Jacobian. The variables $\mathsf{p} = x_2 - x_1$, $\mathsf{q} = x_4 - x_1$ and $\mathsf{r} = x_2 - x_1 + x_4 - x_3$ are used to compute the Jacobian $|F'_K| = {}^{s \wedge t}/_4$: $s = p - \xi_1 r$ and $t = q - \xi_2 r$, where $\xi_1 = \mathsf{qxi}$ and $\xi_2 = \mathsf{qyi}$ are the current coordinates in the reference element.

```
void dg_BDiffusion::integrateEdgeA_(const dg_HP2d001& elmX, const uint i,
                                    const dg_HP2d001& elmY, const uint j,
                                    spc_ElementMatrix<real>& em,
                                    const real signa, const uint relation) {

  // quadrature points
  real qxi = 0.0, qyi = 0.0, qxj = 0.0, qyj = 0.0;

  // Jacobian
  real dx;

  // the derivatives wrt. \xi and x
  Real2d s, t, u, v;

  // pointer into the arrays of the shape functions
  real* bk;
  Real2d* blx;

  // p, q and r are used to calculate the Jacobian
  Real2d p, q, r;
  pqr_(elmY, p, q, r);

  // calculate the displacement and ratio of one edge wrt. the other
  real adjust1, adjust2;
  Real2d e;
  getAdjust_(elmX, elmY, i, j, relation, e, adjust1, adjust2);

  // the outer unit normal vector of elmX
  Unit2d n(e.y(), -e.x());
  DPL(dgBDiffIntegrateEdge_D, "[" << __FILE__ << ", line " << __LINE__
      << "] dg_BDiffusion::integrateEdgeA_ -- n =", n);

  // allocate space for the shape functions
  uint m = elmX.T().n();
  if (m < elmY.T().n())
    m = elmY.T().n();
  if (m > n_) {
    delete[] shpfnc_;
  delete[] shpfncD_;
  shpfnc_ = new real[n_ = m];
  shpfncD_ = new Real2d[n_];
}

DPL(dgBDiffIntegrateEdge_D, "[" << __FILE__ << ", line "
    << __LINE__ << "] dg_BDiffusion::integrateEdgeA_ -- adjust: add "
    << adjust1 << ", mult. by", adjust2);

// the weights and points for the quadrature; quadrature loop
uint gauss;
if (gauss_ == 0) {
  gauss = elmX.p();
  if (gauss < elmY.p())
    gauss = elmY.p();
} else
  gauss = gauss_;
const real* awi = int_GaussAbscWght[gauss];
for (uint qi = 0; qi < gauss + 1; ++qi) {

  qxi = qxj = awi[0];

  if (relation == 2) {
    qxi = adjust1 + qxi*adjust2;
  } else {
    if (relation == 3) {
      qxj = adjust1 + qxj*adjust2;
    }
  }

  // map the quadratur points from 1D to an edge in 2D
  edgeQuadPoint_(i, qxi, qyi);
  edgeQuadPoint_(j, qxj, qyj);

  s.lincomb(p, r, 1.0, −qxi);
  t.lincomb(q, r, 1.0, −qyi);

  // the Jacobian and other integration constants
  // awi[1] is the weight for the numerical integration
  dx = e.l2() / (s^t);
  if (dx < 0.0)
    dx = −dx;
  DPL(dgBDiffIntegrateEdge_D, "[" << __FILE__ << ", line "
      << __LINE__ << "] dg_BDiffusion::integrateEdgeA_ -- dx =", dx);
  dx *= awi[1] * signa * a_;

  u = Real2d(t.y(), −s.y()); u *= dx;
  v = Real2d(−t.x(), s.x()); v *= dx;

  // evaluate the shape functions and their derivatives
```

```
    elmX.evaluate(2.0 * qxi - 1.0, 2.0 * qyi - 1.0, shpfnc_);
    elmY.evaluateD(2.0 * qxj - 1.0, 2.0 * qyj - 1.0, shpfncD_);

    // compute the entries in the element matrix, ie. loop over all shape fcts.
    bk = shpfnc_;
    for (uint k = 0; k < elmX.T().n(); ++k) {
      blx = shpfncD_;

      for (uint l = 0; l < elmY.T().n(); ++l) {
        Real2d blxi( (*blx) * u, (*blx) * v );
        em(k, l) += *bk * ( n * blxi );
        blx++;
      }

      bk++;
    }

    // next quadrature point
    awi += 2;
  } // for qi
}
```

A.2 Boundary Conditions

The declaration and implementation of dg_BoundaryCond are shown in the next two code sections.

xy_Function is simply a wrapper around the formula parser and the evaluation for a parsed formula. These subroutines are included in the tool box of Concepts.

dgBoundary.h

```
/* -*- c++ -*-
 * boundary conditions for DG FEM
 * can possibly be useful for other types of FEM too
 */

#ifndef dgBoundary_h
#define dgBoundary_h

#include "../concepts.h"
#include "../geometry/geoTopology.h"
#include "dgDebug.h"

// ******************************************************* xyFunction **

/** Class to handle an arbitrary 2D function.
 */
```

```
class xyFunction {
  friend ostream& operator<<(ostream& os, const xyFunction& fnc);

public:
  /** Constructor. Parses the formula and saves it in a precompiled
      form. */
  xyFunction(const char* formula);

  /** Destructor. Frees the space used by the parsed formula. */
  ~xyFunction();

  /** Computes the value of the function at the point x. */
  inline const real operator()(const Real2d& x) const;

private:
  /// The parsed formula
  uchar* pgm_;
};

const real xyFunction::operator()(const Real2d& x) const {

  real f = 0.0;
  process(pgm_, x.x(), x.y(), 0.0, &f);
  return f;
}

// ********************************************************** dg_Boundary **

/** Class to describe the boundary condition of an element of the
    topology. A boundary condition of type Neumann or Dirichlet
    consist of the type and a function. These two properties can
    be requested.

    Boundary conditions of type Robin are not yet implemented.

    This class is implemented to serve the needs of DG FEM but it can
    possibly be useful for hpFEM too.

    @author Philipp Frauenfelder
 */
class dg_BoundaryCond : public geo_Attribute {

public:
  /// The different boundary condition types
  enum boundaryTypes { FREE = 0, DIRICHLET, NEUMANN, MAX_TYPE };

  /** Constructor. The type of the boundary condition must be one of
      FREE, DIRICHLET or NEUMANN. If it's FREE, the formula is of no
      use and can be omitted. If any other boundary condition is given
      without formula, then "(0)" is assumed. */
  dg_BoundaryCond(uint attrib, const char* formula = 0);

  /** Returns the type of the boundary condition */
  inline const uint getType() const;

  /** Application operator. Calculates the value of the boundary
      function at a specific point. */
  inline const real operator()(const Real2d& x) const;

  /// Returns information about itself
  virtual ostream& info(ostream& os) const;

private:
  /// The function of the boundary condition
  xyFunction* fnc_;
};

const uint dg_BoundaryCond::getType() const {

  return attrib();
}

const real dg_BoundaryCond::operator()(const Real2d& x) const {

  if (fnc_) {
    return (*fnc_)(x);
  } else {
    return 0.0;
  }
}

#endif // dgBoundary_h
```

dgBoundary.cc

```
/* dg-Boundary
 */

#include "../concepts.h"
#include "dgDebug.h"
#include "dgBoundary.h"
#include "../toolbox/tbxFuCo.h"

// ********************************************************** xyFunction **

xyFunction::xyFunction(const char* formula) {

  uchar pgm[FuCo_MaxPgmSize];
  uint  len;
  if (!(len = parse(formula, pgm))) {
                                                     // *** exception ***
    cerr << '[' << __FILE__ << ", line " << __LINE__;
    cerr << "] xyFunction::xyFunction() -- formula syntax error\n";
    exit(1);
  } else {
    pgm_ = new uchar[len];
    memcpy(pgm_, pgm, len * sizeof(pgm_[0]));
  }
}

xyFunction::~xyFunction() {
  delete[] pgm_;
}

ostream& operator<<(ostream& os, const xyFunction& fnc) {
  return os << "xyFunction";
}

// ******************************************************** dg_Boundary **

dg_BoundaryCond::dg_BoundaryCond(uint attrib, const char* formula) :
  geo_Attribute(attrib), fnc_(0) {

  assert( attrib < MAX_TYPE );

  if ( (attrib != FREE) && (formula) ) {
    fnc_ = new xyFunction(formula);
  }
}

ostream& dg_BoundaryCond::info(ostream& os) const {

  os << "dg_BoundaryCond(" << attrib() << ", ";
  if (fnc_ == 0)
    os << "0";
  else
    os << *fnc_;
  return os << ")";
}
```

A.3 Sort of Adaptivity

The next code snippet shows the implementation of the adaptive algorithm 4.7. It simply replaces the refinement code in the main program (c. f. section A.6).

The parameter eta determines if an element is refined or not, depending on its local error. If the local error is larger than (eta · maximal error) the element is refined (i. e. it is subdivided or the polynomial degree is raised).

```
  // ********************************************************************
  // Phase 3: refining
  if (i > 0) {
    // for the L shaped domain (geo_LShape):
#define NR_NODES 6
    Real2d nodes[NR_NODES] = {Real2d( 0.0, 0.0),
                              Real2d( 1.0, 0.0),
                              Real2d( 1.0, 1.0),
                              Real2d(-1.0, 1.0),
                              Real2d(-1.0,-1.0),
                              Real2d( 0.0,-1.0)};

    cout << " Refining (adapt.): " << flush;
    // get maximal error
    sc = spc.scan();                            cout << "." << flush;
    real maxErr = 0.0;
    while (*sc) {
      dg_HP2d001& elm = (dg_HP2d001&)(*sc)++;
      real err = fabs(elm.getError());
      if (err > maxErr)
        maxErr = err;
    }
    delete sc;                                  cout << "." << flush;

    // refine elements
    sc = spc.scan();                            cout << "." << flush;
    maxErr *= eta;
    while (*sc) {
      dg_HP2d001& elm = (dg_HP2d001&)(*sc)++;
      if (fabs(elm.getError()) > maxErr) {
        bool subdivide = false;
        for (uint j = 0; j < NR_NODES; j++)
          subdivide |= elm.pointInElement(nodes[j]);
        if (subdivide) {
          spc.adjust(elm, 1, 0);                cout << 'l' << flush;
        } else {
          spc.adjust(elm, 0, 1);                cout << 'p' << flush;
        }
      }
    }
                                                cout << "." << flush;
    delete sc;                            cout << " done: " << spc << endl;
```

A.4 Debugging Techniques

The debugging techniques described in section 4.3.4 need some header files. The top level header file `debug.h` and one of header files in the packages are shown here.

`debug.h`

This is the top level debugging header file.

```
#ifndef debug_h
#define debug_h

#ifdef DEBUG

#include<stream.h>

/// Debug Print Line
#define DPL(doit, msg, var)\
if(doit!=0) cout << msg << " " << var << endl;

/// Debug Print
#define DP(doit, msg, var)\
if(doit!=0) cout << msg << " " << var;

#else
 #define DP(doit, msg, var)
 #define DPL(doit, msg, var)
 #define NDEBUG // discard assertions
#endif

#include <assert.h>
#include <iomanip>

#endif
```

`opDebug.h`

This is the debugging header file of the operator package.

```
#ifndef opDebug_h
#define opDebug_h

#include "../debug.h"

#ifdef DEBUG

// debugging opDGESV
#define opDGESVConstr_D 0
```

```
#define opDGESVAppl_D 0

#endif

#endif
```

A.5 Mesh Generation on the L Shaped Domain

The following two files contain the code for the mesh generation for the L shaped domain (c. f. figure 5.7 on page 52). The declaration of the classes for the L shaped domain and its mesh and the scanner for the mesh of the L shaped domain are in the file `lshape.h`. The implementation of the constructor and the destructor of `dg_LShape` are in the file `lshape.cc`.

`lshape.h`

```
/* -*- c++ -*-
 * L shaped domain
 */

#ifndef lshape_h
#define lshape_h

#include <typeinfo>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>

#include "concepts.h"
#include "geometry.h"
#include "dg.h"

/** Three unit quadrilaterals forming an L shaped domain (third quadrant
    missing).
*/
class dg_LShape : public geo_Mesh2 {

public:
  /** Constructor. Creates the vertices, edges and quadrilaterals
      in the topology and arranges them with their element map in cells.
      The boundary conditions are intialized and assigned to the edges
  */
  dg_LShape();
  virtual ~dg_LShape();

  /// Returns the number of cells in the mesh
  inline uint ncell() const {
    return 3;
  }

  /// Returns a scanner for the mesh of the domain
  inline geo_Scan2* scan() {
    return new S(cell_);
  }

private:
  /** Subclass of dg_LShape used to scan the mesh of the square
      @see dg_LShape
  */
  class S : public tbx_Scan<geo_Cell2> {

  public:
    inline S(geo_Quad2d *(&cell)[3]) : idx_(0), cell_(cell) {}
    inline S(const S& scan) : idx_(scan.idx_), cell_(scan.cell_) {}

    inline bool eos() const {
      return idx_ == 3;
    }

    inline geo_Cell2& operator++(int) {
      return *cell_[idx_++];
    }

    inline geo_Scan2* clone() const {
      return new S(*this);
    }

  private:
    uint        idx_;
    geo_Quad2d  *(&cell_)[3];
  };

  /// The vertices of the L shaped domain
  geo_Vertex   *vtx_[8];

  /// The edges of the L shaped domain
  dg_Edge      *edg_[10];

  /// The quadrilaterals of the L shaped domain
  dg_Quad      *quad_[3];

  /// The cells in the mesh
  geo_Quad2d *cell_[3];

  /// Boundary conditions
  dg_BoundaryCond *dirichlet1_, *dirichlet2_, *neumann1_, *neumann2_;
```

```
};

#endif
```

lshape.cc

```
/* L shaped domain
 */

#include "lshape.h"
#include "dg.h"

dg_LShape::dg_LShape() {

  dirichlet1_ = new dg_BoundaryCond(dg_BoundaryCond::DIRICHLET, "(sqrt(x))");
  dirichlet2_ = new dg_BoundaryCond(dg_BoundaryCond::DIRICHLET, "(sqrt(-y))");
  neumann1_ = new dg_BoundaryCond(dg_BoundaryCond::NEUMANN,
                                  "(1/(2*((1+y*y)^(3/4))))");
  neumann2_ = new dg_BoundaryCond(dg_BoundaryCond::NEUMANN,
                                  "(1/(2*((1+x*x)^(3/4))))");

  for (uint i = 0; i < 8; ++i) {
    vtx_[i] = new geo_Vertex();
  }

  edg_[0] = new dg_Edge(*vtx_[0], *vtx_[1], neumann2_);
  edg_[1] = new dg_Edge(*vtx_[1], *vtx_[2], dirichlet2_);
  edg_[2] = new dg_Edge(*vtx_[2], *vtx_[3]);
  edg_[3] = new dg_Edge(*vtx_[3], *vtx_[0], neumann1_);
  edg_[4] = new dg_Edge(*vtx_[2], *vtx_[4]);
  edg_[5] = new dg_Edge(*vtx_[4], *vtx_[5], neumann2_);
  edg_[6] = new dg_Edge(*vtx_[5], *vtx_[3], neumann1_);
  edg_[7] = new dg_Edge(*vtx_[6], *vtx_[2], dirichlet1_);
  edg_[8] = new dg_Edge(*vtx_[6], *vtx_[7], neumann1_);
  edg_[9] = new dg_Edge(*vtx_[7], *vtx_[4], neumann2_);

  quad_[0] = new dg_Quad(*edg_[0], *edg_[1], *edg_[2], *edg_[3]);
  quad_[1] = new dg_Quad(*edg_[2], *edg_[4], *edg_[5], *edg_[6]);
  quad_[2] = new dg_Quad(*edg_[7], *edg_[8], *edg_[9], *edg_[4]);

  // bottom left
  cell_[0] = new geo_Quad2d(*quad_[0], geo_MapQuad2d("(x-1, y-1)", 1.0, 1.0));
  // top left
  cell_[1] = new geo_Quad2d(*quad_[1], geo_MapQuad2d("(x-1, y)", 1.0, 1.0));
  // top right
  cell_[2] = new geo_Quad2d(*quad_[2], geo_MapQuad2d("(x, y)", 1.0, 1.0));
}

dg_LShape::~dg_LShape() {
  for(uint l = 2; l--;) delete cell_[l];
  for(uint k = 2; k--;) delete quad_[k];
  for(uint j = 9; j--;) delete edg_[j];
  for(uint i = 7; i--;) delete vtx_[i];

  delete neumann1_;
  delete neumann2_;
  delete dirichlet1_;
  delete dirichlet2_;
}
```

A.6 Main program

The file `dgfem.cc` contains the main program. The comments follow the steps in section 3.3, where the main steps in a Concepts application are listed.

After the source code, the output of the program `dgfem` is shown.

```
/* DG FEM
 */

#include <typeinfo>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>
#include <unistd.h>

#include "concepts.h"
#include "function.h"
#include "operator.h"
#include "hp.h"
#include "dg.h"
#include "lshape.h"

int main(int argc, char** argv) {

  uint p = 1, gauss = 0, depth = 1;
  real mu = 1.0, stabilization = 0.0;
  uint graphicpoints = 20;
  bool debug = false;

  ofstream *ofs, *errfs;
  int opt;
  dg_HP2dScan* sc;

  cout.setf(ios::scientific, ios::floatfield);
  cout.setf(ios::showpos);
cout.precision(3);

// *************************************************************************
// parsing command line options
while ((opt = getopt(argc, argv, "-m:p:g:s:dD:G:")) != EOF)
  switch(opt) {
  case 'm':
    mu = atof(optarg);                              break;
  case 'p':
    p = atoi(optarg);                               break;
  case 'g':
    gauss = atoi(optarg);                           break;
  case 's':
    stabilization = atof(optarg);                   break;
  case 'd':
    debug = true;                                   break;
  case 'D':
    depth = atoi(optarg);                           break;
  case 'G':
    graphicpoints = atoi(optarg);                   break;
  default:
    cout << "Call: " << argv[0]
         << " [-p DEGREE] [-g GAUSS] [-D DEPTH] [-m MU] "
         << "[-s STAB] [-G GPOINTS] [-d]" << endl
         << "where" << endl
         << " DEGREE: polynomial degree" << endl
         << " GAUSS: number of quadrature points (0: as much as needed)"
         << endl
         << " DEPTH: maximal level" << endl
         << " MU: slope of linear polynomial degree distribution vector"
         << endl
         << " STAB: stabilization coefficient" << endl
         << " GPOINTS: number of points in the graphics in each directions"
         << endl
         << " -d: debug, ie. print the matrices" << endl;
    exit(1);
  }

cout << '[' << argv[0] << "]" << endl;
cout << "--" << endl << "Parameters: " << endl
     << "degree = " << p << endl << "gauss = " << gauss << endl
     << "depth = " << depth << endl << "mu = " << mu << endl
     << "stab = " << stabilization << endl
     << "gpoints = " << graphicpoints << endl;

// *************************************************************************
// problem: - a Delta u + c u = f
const real a(1.0);
const real c(1.0);
```

```
  // exact solution
  const char* uex = "(sqrt(sqrt(x*x+y*y)))";

  // RHS
  const char* fex =
    "(3*x*x/(4*((x*x+y*y)^(7/4)))+3*y*y/(4*((x*x+y*y)^(7/4)))-"
    "1/((x*x+y*y)^(3/4))+((x*x+y*y)^(1/4)))";

  cout << "--" << endl << "Problem: "
       << "- a Delta u + c u = f" << endl
       << "a = " << a << endl << "c = " << c << endl
       << "u = " << uex << endl << "f = " << fex << endl;

  // ************************************************************************
  // Phases 1 and 2: create mesh and space
  cout << "--" << endl << "Mesh and space: " << flush;
  dg_LShape msh;                                     cout << "." << flush;
  geo_Bool bc;                                       cout << "." << flush;
  dg_HP2d spc(msh, 0, p, &bc);           cout << " done: " << spc << endl;

  // graphic of the mesh
  ofs = new ofstream("dg0.eps");
  spc.sketch(*ofs, 100);
  delete ofs;


  // ************************************************************************
  // computations
  cout << "--" << endl << "Computations:" << endl;
  errfs = new ofstream("error.data");

  for (uint i = 0; i <= depth; ++i) {

    cout << endl << "Iteration " << i << endl;

    // **********************************************************************
    // Phase 3: refining
    if (i > 0) {
      // vertices of the L shaped domain (dg_LShape):
#define NR_NODES 6
      Real2d nodes[NR_NODES] = {Real2d( 0.0, 0.0),
                                Real2d( 1.0, 0.0),
                                Real2d( 1.0, 1.0),
                                Real2d(−1.0, 1.0),
                                Real2d(−1.0,−1.0),
                                Real2d( 0.0,−1.0)};

      cout << " Refining (lin. deg. vec.): " << flush;
      uint l = 0; // max level
  // get max level
  sc = spc.scan();                              cout << "." << flush;
  while (*sc) {
    dg_HP2d001& elm = (dg_HP2d001&)(*sc)++;

    const dg_Quad* quad = dynamic_cast<const dg_Quad*>(&elm.support());
    if (quad)
      if (l < quad->level())
        l = quad->level();
  }
  l++;
  cout << '.' << flush;
  delete sc;

  // refine elements
  sc = spc.scan();                              cout << "." << flush;
  while (*sc) {
    dg_HP2d001& elm = (dg_HP2d001&)(*sc)++;

    bool subdivide = false;
    for (uint j = 0; j < NR_NODES; j++)
      subdivide |= elm.pointInElement(nodes[j]);
    if (subdivide) {
      spc.adjust(elm, 1, 0);                    cout << 'l' << flush;
    } else {
      const dg_Quad* quad = dynamic_cast<const dg_Quad*>(&elm.support());
      if (quad) {
        spc.adjust(elm, 0, (int)floor((real)(1+l-quad->level())*mu)
                   - elm.p());                  cout << 'p' << flush;
      }
    }
  }
                                                cout << "." << flush;
  delete sc;                              cout << " done: " << spc << endl;

  if (i == depth) {
    // graphic of the mesh
    cout << " Graphic of the mesh." << endl;
    ofs = new ofstream("dg1.eps");
    spc.sketch(*ofs, 100);
    delete ofs;
  }
}

// ************************************************************************
// Phase 4: create the differential operators and their matrices
cout << " Stiffness matrix: " << flush;
dg_BDiffusion bf(gauss, a);                     cout << "." << flush;
op_Local<real> A1(spc, spc, bf);                cout << "." << flush;
```

```
dg_Identity id(gauss);                              cout << "." << flush;
op_Local<real> I(spc, spc, id);                     cout << "." << flush;

dg_BDiscont st(gauss, stabilization);               cout << "." << flush;
op_Local<real> S(spc, spc, st);                     cout << "." << flush;

op_LiCo<real> LL(A1, I, 1.0, c);                    cout << "." << flush;
op_LiCo<real> L(LL, S, 1.0, 1.0);               cout << " done." << endl;

if (debug) {
  cout << " Diffusion matrix: " << A1 << endl
       << " Stabilization matrix: " << S << endl
       << " Identity matrix: " << I << endl;
}

// ************************************************************************
// Phase 5: the linear forms and the vectors of the right hand side
cout << " Load vector: " << flush;
dg_LDiffusion ldiff(gauss, a);                      cout << "." << flush;
fnc_Vector<real> fdiff(spc, ldiff);                 cout << "." << flush;

hp_Riesz lf(fex, gauss);                            cout << "." << flush;
fnc_Vector<real> ff(spc, lf);                       cout << "." << flush;

dg_LDiscont ldis(gauss, stabilization);             cout << "." << flush;
fnc_Vector<real> fdis(spc, ldis);                   cout << "." << flush;

ff += fdiff;                                        cout << "." << flush;
ff += fdis;                                     cout << " done." << endl;

// ************************************************************************
// Phase 6: solving the system
cout << " Solving: " << flush;
fnc_Vector<real> u(spc);                            cout << "." << flush;
op_DGESV Linv(L);                                   cout << "." << flush;
Linv(ff, u);                              cout << " done: " << Linv << endl;

// ************************************************************************
// exact solution
cout << " Exact solution: " << flush;
hp_Riesz lf_u(uex, gauss);                          cout << "." << flush;
fnc_Vector<real> uu(spc, lf_u);                     cout << "." << flush;
fnc_Vector<real> ue(spc);                           cout << "." << flush;
op_DGESV Iinv(I);                                   cout << "." << flush;
Iinv(uu, ue);                                   cout << " done." << endl;

if (debug) {
  cout << " Right hand side: " << endl << ff << endl
       << " Diffusion part of RHS: " << endl << fdiff << endl
       << " Stabilization of RHS: " << endl << fdis << endl
```

```
          << " Solution: " << endl << u << endl
          << " Exact Solution (by L^2 projection): " << endl << ue << endl;
    }

    // ***********************************************************************
    // Phase 7: error estimation
    fnc_Vector<real> diff(ue);
    diff −= u;

    if (i == depth) {
      // *********************************************************************
      // Phase 8: postprocessing
      cout << " Save data for gnuplot to disk." << endl;
      spc.storeSolution(ue);
      ofs = new ofstream("dg0exact.data");
      spc.drawGraph(*ofs, graphicpoints);
      delete ofs;
      spc.storeSolution(u);
      ofs = new ofstream("dg0.data");
      spc.drawGraph(*ofs, graphicpoints);
      delete ofs;
      spc.storeSolution(diff);
      ofs = new ofstream("dg0error.data");
      spc.drawGraph(*ofs, graphicpoints);
      delete ofs;
    } else {
      spc.storeSolution(diff);
    }

    // L^2 error
    I(diff, ff);
    spc.storeWeights(ff);
    *errfs << spc.dim() << " " << spc.getError() << " ";
    cout << " ||u-u_(l,p)||_2^2 = " << spc.getError() << endl;

    // energy error
    L(diff, ff);
    spc.storeWeights(ff);
    *errfs << spc.getError() << " ";
    cout << " ||u-u_(l,p)||_E^2 = " << spc.getError() << endl;
    *errfs << Linv.condition() << endl;
  }
  delete errfs;

  // *************************************************************************
  // Phase 9: removal of the matrices, vectors, the space and the mesh
  return 0;
}
```

Output of the Main Program

```
[dgfem]
--
Parameters:
degree  = 1
gauss   = 0
depth   = 1
mu      = +1.000e+00
stab    = +0.000e+00
gpoints = 20
--
Problem: - a Delta u + c u = f
a = +1.000e+00
c = +1.000e+00
u = (sqrt(sqrt(x*x+y*y)))
f = (3*x*x/(4*((x*x+y*y)^(7/4)))+3*y*y/(4*((x*x+y*y)^(7/4)))
    -1/((x*x+y*y)^(3/4))+((x*x+y*y)^(1/4)))
--
Mesh and space: .. done: dg_HP2d(dim = 12, nelm = 3)
--
Computations:

Iteration 0
  Stiffness matrix: ....... done.
  Load vector: ....... done.
  Solving: .. done: op_DGESV(dim = 12, cond = +8.745e+00, factorized)
  Exact solution: .... done.
  ||u-u_(l,p)||_2^2 = +3.543e-02
  ||u-u_(l,p)||_E^2 = +1.509e-01

Iteration 1
  Refining (lin. deg. vec.): ...lll. done: dg_HP2d(dim = 48, nelm = 12)
  Graphic of the mesh.
  Stiffness matrix: ....... done.
  Load vector: ....... done.
  Solving: .. done: op_DGESV(dim = 48, cond = +3.200e+01, factorized)
  Exact solution: .... done.
  Save data for gnuplot to disk.
  ||u-u_(l,p)||_2^2 = +2.964e-02
  ||u-u_(l,p)||_E^2 = +1.340e-01
```

Symbol Index

$[v]$	jump of v over an edge
$\lvert F_K' \rvert$	Jacobian of the element map F_K
$\langle v \rangle$	average of v over an edge
$\mathcal{E}$	set of smallest element edges
$\mathcal{E}_{\text{int}}$	set of interior element edges
$\mathcal{P}_p(\hat{\Omega})$	polynomials of total degree p on $\hat{\Omega}$
$\{\varphi_i^K\}_{i=1}^{N_K}$	shape functions on element K
$\{\varphi_i\}_{i=1}^{N}$	basis of the (DG)FE-space
$\mathcal{T}$	partition of Ω, a FE-mesh
δ_K	stabilisation parameter
$\lfloor v \rfloor$	oriented jump of v over an edge
Γ_+	outflow boundary
Γ_-	inflow boundary
Γ_0	diffusion boundary
Γ_{int}	union of the set of interior element edges
$\hat{\Omega}$	reference element $(0,1)^2$
$\partial_- K$	element inflow boundary
$\underline{n}_e$	numbering dependent unit normal vector of an edge
$\underline{n}_K$	unit outward normal vector of element K
$\underline{p}$	degree vector $\{p_K : K \in \mathcal{T}\}$
ξ	coordinates in the reference element $\hat{\Omega}$
e	edge of an element of $\mathcal{T}$
F_K	element map: $F_K : \hat{\Omega} \to K$
h_K	diameter of $K \in \mathcal{T}$
K	open and connected element of $\mathcal{T}$
N	dimension of the (DG)FE-space
$N_i(\xi)$	shape function on the reference element $\hat{\Omega}$

N_K	number of shape functions on element K
p_K	polynomial degree on element K
v^+	inner trace of v
v^-	outer trace of v
$W^{1,\infty}(\Omega)^{2\times 2}_{\text{sym}}$	space of symmetric 2×2 matrices of Sobolev functions in $W^{1,\infty}(\Omega)$
x^i	i^{th} component of the vector $x \in \mathbb{R}^d$
x_i	i^{th} vertex of an element $K \in \mathcal{T}$
$\mathcal{B}^l_\beta(\Omega)$	countably normed space
$\mathcal{S}^{p,0}_0(\Omega,\mathcal{T})$, $\mathcal{S}^{p}_0(\Omega,\mathcal{T})$	DGFE-space: $\left\{u \in L^2(\Omega) : u\|_K \circ F_K \in \mathcal{P}_{p_K}(\hat{\Omega}),\, u\|_{\partial K\cap\Gamma_D} = 0, \forall K \in \mathcal{T}\right\}$
$\mathcal{S}^{p,1}_0(\Omega,\mathcal{T})$	FE-space: $\left\{u \in H^1_0(\Omega) : u\|_K \circ F_K \in \mathcal{P}_{p_K}(\hat{\Omega}), \forall K \in \mathcal{T}\right\}$
$\hat{\mathcal{T}}^n_\sigma$, $\mathcal{T}^n_\sigma$	geometric mesh family
$H^{m,l}_\beta(\Omega)$	weighted Sobolev space

Bibliography

[1] Free Software Foundation, Inc. [2000], *GCC—The GNU Compiler Collection*, Internet.

The GNU compiler is used to compile Concepts. `http://www.fsf.org/software/gcc/gcc.html`

[2] Netlib [1994], *LAPACK – Linear Algebra PACKage*, Internet.

Large library of linear algebra subroutines in Fortran. A C++ version is also available but was not used for this diploma thesis. `http://www.netlib.org/lapack/`

[3] I. Babuška and B. Q. Guo [1988], *Regularity of the Solutions of Elliptic Problems with Piecewise Analytic Data I*, SIAM J. Math. Anal., 19:172–203.

[4] I. Babuška and B. Q. Guo [1989], *Regularity of the Solutions of Elliptic Problems with Piecewise Analytic Data II*, SIAM J. Math. Anal., 20:763–781.

[5] Timothy J. Barth and Herman Deconinck (Editors) [1999], *High order methods for computational physics*, vol. 9 of *Lecture notes in computational science and engineering*, chap. 6, pp. 365–374, Springer.

Gives a compact derivation of the used DGFEM variational formulation and a short note on stability.

[6] P. Frauenfelder [1999], *Schnellere Quadratur für hp-FEM in drei Dimensionen*, Semester Thesis.

Sum factorization exploiting the tensor product Ansatz of the shape functions during the computation of the element matrices.

[7] Martin Gogolla, *UML for the Impatient*, University of Bremen, FB 3, Computer Science Departement, Postfach 330440, D-28334 Bremen, Germany.

A short introduction to UML by examples.

[8] Nina P. Hancke [1998], *Calculating Large Spectra in Hydrodynamic Stability: a p-FEM Approach to Solve the Orr Sommerfeld Equation*, Master's thesis, Swiss Federal Institute of Technology, CH-8092 Zürich.

A the short and good summary of doubly integrated Legendre polynomials can be found on page 22 and following.

[9] Hewlett-Packard Company, Silicon Graphics Computer Systems, Inc. [1996], *Standard Template Library Programmer's Guide.*

`http://www.sgi.com/Technology/STL/`

[10] Christian Lage [1995], *Softwareentwicklung zur Randelementmehtode: Analyse und Entwurf effizienter Techniken*, Ph.D. thesis, Christian-Albrechts-Universität, Kiel.

First ideas of Concepts.

[11] Christian Lage [1998], *Concept Oriented Design of Numerical Software*, Tech. Rep. 98-07, Swiss Federal Institute of Technology, CH-8092 Zürich.

Abstract presentation of the ideas behind Concepts by its author.

[12] Rational Corporation, Santa Clara [1997], *Object Constraint Language (Version 1.1).* `http://www.rational.com/uml/resources/documentation/ocl/`

[13] Rational Corporation, Santa Clara [1997], *UML Notation Guide (Version 1.1).* `http://www.rational.com/uml/resources/documentation/notation/`

[14] Rational Corporation, Santa Clara [1997], *UML Semantics (Version 1.1).* `http://www.rational.com/uml/resources/documentation/semantics/`

[15] A. Rüegg and D. Hoch [1999], *FEM für elliptische Probleme mit dem Programmsystem Concepts-1.4*, Semester Thesis.

FEM with mixed boundary values in Concepts. Features a short introduction into the main parts of Concepts.

[16] Bjarne Stroustrup [1997], *The C++ Programming Language*, Addison Wesley Longman, Inc., 3rd edn.

The reference for C++.

[17] E. Süli, P. Houston and C. Schwab [1999], *hp-Finite Element Methods for Hyperbolic Problems*, Tech. Rep. 99-14, Swiss Federal Institute of Technology, CH-8092 Zürich.

[18] T. P. Wihler and C. Schwab [2000], *Exponential convergence of the hp-DGFEM for Diffusion Problems in two Space Dimensions.*

Theoretical background for the computations in this thesis.

[19] Thomas Williams and Colin Kelley [1999], *Gnuplot Central*, Internet.

Gnuplot was used to create the graphs in this thesis. `http://www.ucc.ie/gnuplot/`

[20] Andreas Zeller et al. [2000], *DDD—Data Display Debugger*, Internet.

One of the greatest debuggers with a nice, graphical user interface. Besides usual front-end features such as viewing source texts, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs. `http://www.gnu.org/software/ddd/`