Unterschrift des Betreuers

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

A posteriori Fehlerschätzer für die Symmsche Integralgleichung in 3D

Ausgeführt am Institut für
Analysis und Scientific Computing
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.Math. Dr.techn. Dirk Praetorius

durch
Samuel Ferraz-Leite
Haberlgasse 95/11
1160 Wien

Datum

Unterschrift

Kurzfassung

Nur für die wenigsten partiellen Differentialgleichungen lassen sich Lösungen in analytisch geschlossener Form schreiben und quantitativ auswerten. Daher interessiert man sich in der Praxis für numerische Verfahren, die die exakte Lösung auf eine gewisse Genauigkeit hin approximieren.

Die Kontrolle des Approximationsfehlers in numerischen Simulationen erfordert die Herleitung von sogenannten a posteriori Fehlerschätzern, die den Fehler in einer problemabhängigen Norm kontrollieren. Um ein praktisch berechenbares Maß des Fehlers zu erhalten, dürfen diese Fehlerschätzer lediglich von den Problemdaten und bereits berechneten diskreten Lösungen abhängen, nicht aber von der im Allgemeinen unbekannten exakten Lösung.

Neben der Kontrolle des Approximationsfehlers dienen a posteriori Fehlerschätzer auch zur Steuerung effizienter Diskretisierungsstrategien: In numerischen Simulationen liegt ein Hauptaugenmerk auf der effizienten Nutzung von Systemressourcen. Dazu ist eine Diskretisierung erforderlich, die die wesentliche Anzahl an Freiheitsgraden zur Approximation von möglichen Singularitäten der exakten Lösung einsetzt, während Bereiche, in denen die Lösung glatt ist, möglichst wenig aufgelöst werden. Dazu müssen numerische Algorithmen Kriterien beinhalten, nach denen sie das lokale Verhalten der Lösung beurteilen.

Für elliptische Differentialgleichungen ist neben der wohlbekannten Finite Elemente Methode (FEM) die sogenannte Randelementmethode (BEM, engl. *boundary element method*) eine alternative Diskretisierungsstrategie. In der BEM wird die partielle Differentialgleichung mithilfe der Fundamentallösung in eine Integralgleichung überführt. Im Fall der direkten Randintegralmethode erhält man so eine elliptische Integralgleichung der ersten Art, auf die das Galerkinverfahren angewendet werden kann.

Anders als in der FEM ist die a posteriori Fehlerschätzung in der BEM wesentlich weniger gut verstanden. Diese Arbeit liefert einen Beitrag zur mathematischen Herleitung von a posteriori Fehlerschätzern für die Galerkin-BEM der schwach singulären Integralgleichung zum Laplace-Operator. Diese sogenannte Symmsche Integralgleichung ist äquivalent zum Laplace-Problem mit reinen Dirichlet-Randbedingungen. Die erzielten Resultate übertragen sich aber wörtlich von der Laplace- auch auf die Lameé- und die Stokes-Gleichung.

Im Wesentlichen betrachten wir zwei Strategien zur a posteriori Fehlerschätzung: Zum einen die klassische $h - h/2$-Strategie, zum anderen eine Glättungsstrategie. Bei der $h - h/2$-Strategie berechnet man neben der Lösung ϕ_h zu einer gegebenen Triangulierung $\mathcal{T}_h$ des Randes auch die Lösung $\phi_{h/2}$ zur uniformen Verfeinerung von $\mathcal{T}_h$. Die Energienorm $\eta_H = \|\phi_{h/2} - \phi_h\|$ der Differenz beider diskreter Lösungen kann dann als Fehlerschätzer dienen. Wir beweisen die Effizienz des Fehlerschätzers η_H in dem Sinn, dass stets $\eta_H \leq \|\phi - \phi_h\|$ gilt, wobei ϕ die unbekannte exakte Lösung bezeichne. Insbesondere impliziert die Konvergenz des Verfahrens also die Konvergenz von η_H mit mindestens derselben Ordnung. Unter der Saturationsannahme beweisen wir die Existenz einer Konstante $C > 0$ mit $\|\phi - \phi_h\| \leq C\eta_H$. Eine solche Abschätzung bezeichnet man als Zuverlässigkeit, da die Konvergenz des Schätzers gegen Null, die Konvergenz des Verfahrens impliziert. In numerischen Beispielen wird die Saturationsannahme experimentell überprüft und bestätigt.

Lokalisierungstechniken mittels gewichteter L^2-Normen erlauben die mathematische Herleitung von lokalen Verfeinerungsindikatoren zur Steuerung einer adaptiven Netzverfeinerung. Das bisherige Fehlen solcher Lokalisierungstechniken ist offenbar der Grund, dass dieser elementare und im Kontext gewöhnlicher Differentialgleichungen wohlbekannte Ansatz in der Literatur zur BEM bisher noch nicht berücksichtigt worden ist.

Bei der Darstellung der Glättungstechnik folgen wir der Arbeit Carstensen/Praetorius 2006 [8], die unter vergleichsweise starken Voraussetzungen die Zuverlässigkeit und Effizienz des Glättungsschätzers η_M zur Fehlerschätzung von $\|\phi - \phi_{h/2}\|$ zeigt. Ein überraschendes Resultat und ein neuer Beitrag dieser Arbeit ist, dass η_M stets äquivalent zum $h - h/2$-Fehlerschätzer η_H ist. Dies impliziert nämlich eine wesentliche Abschwächung der Voraussetzungen zum Ergebnis in Carstensen/Praetorius 2006 [8].

Gewidmet meinem Sohn Joel

Inhaltsverzeichnis

1 Einleitung und Modellproblem **1**
1.1 Modellproblem . . . 1
1.2 Motivation und Kernresultate . . . 3
1.3 Aufbau der Arbeit . . . 4
1.4 Danksagung . . . 6

2 Die Randelementmethode **7**
2.1 Analytische Grundlagen . . . 7
2.1.1 Sobolevräume . . . 7
2.1.2 Sätze aus der Analysis . . . 9
2.2 Variationsformulierung des Modellproblems . . . 11
2.3 Triangulierung . . . 12
2.4 Unstetige Randelemente . . . 16
2.5 Die Galerkin-Randelementmethode . . . 16
2.6 Einfluss von Störungen . . . 19
2.7 A priori Fehlerschätzung . . . 21

3 A posteriori Fehlerschätzung **25**
3.1 Lokalisierungstechniken . . . 26
3.2 Der $h - h/2$-Schätzer . . . 28
3.2.1 Lokalisierung des $h - h/2$-Schätzers . . . 29
3.3 Glättungstechniken . . . 30
3.3.1 Abstrakte Analysis . . . 30
3.3.2 η_M für die Symmsche Integralgleichung . . . 33
3.3.3 Lokalisierung von η_M . . . 33
3.4 Vergleich zwischen η_H und η_M . . . 35

4 Die Programmbibliothek FLBEM **39**
4.1 Einfache Datenstrukturen . . . 39
4.2 Triangulierung . . . 42
4.3 Integraloperatoren . . . 48

5 Numerische Experimente **55**
5.1 Adaptive Algorithmen . . . 55
5.1.1 Adaptiver isotroper Algorithmus . . . 55
5.1.2 Anisotrope Strategie . . . 57
5.2 Kapazität des Referenzwürfels $(-1, 1)^3$ . . . 58
5.3 Schirmproblem auf L-förmigem Randstück . . . 71

A Quelltexte **83**
A.1 Das Programm *symm_mu_H_estimator* 83
A.2 Das Programm *symm_mu_M_estimator* 88
A.3 Triangulierung 95
A.3.1 geometrics.h 95
A.3.2 geometrics.cpp 98
A.3.3 triangulation.h 101
A.3.4 triangulation.cpp 106
A.3.5 triangulationio.h 128
A.3.6 triangulationio.cpp 129
A.3.7 usual_meshes.h 141
A.3.8 usual_meshes.cpp 142
A.4 Integraloperatoren 148
A.4.1 maiprogs.h 148
A.4.2 quadrature.h 148
A.4.3 quadrature.cpp 149
A.4.4 integral_operators.h 174
A.4.5 integral_operators.cpp 176

Literaturverzeichnis **229**

Kapitel 1

Einleitung und Modellproblem

Als Verfahren zur numerischen Lösung elliptischer Randwertprobleme wird üblicherweise die Methode der Finiten Elemente (FEM) verwendet. Bei Kenntnis der Fundamentallösung des Differentialoperators ist es möglich, das Problem in eine Randintegralgleichung zu überführen, deren Diskretisierung auf die sogenannte Randelementmethode (BEM, engl. *boundary element method*) führt. Dieses numerische Verfahren bietet einige Vorteile gegenüber der FEM. Zum einen ist durch die Umformulierung des gegebenen Randwertproblems im $\mathbb{R}^d$ lediglich eine $(d-1)$-dimensionale Mannigfaltigkeit zu diskretisieren. Dies vereinfacht die Netzverwaltung und Netzverfeinerung und verringert die Anzahl der Freiheitsgrade. Zweitens konvergiert die Galerkin-BEM bei fixiertem Polynomgrad mit generisch höherer Ordnung als die FEM. Insgesamt ist also die Problemgröße nach der Diskretisierung im Vergleich zur FEM bei gleicher Genauigkeit wesentlich kleiner. Drittens kann die FEM im Gegensatz zur BEM nur zum Lösen von Gleichungen auf beschränkten Gebieten verwendet werden. Zahlreiche Aufgabenstellungen sind jedoch als Außenraumproblem gestellt. Ein wesentlicher Nachteil der BEM gegenüber der FEM ist, dass die Fundamentallösung des Differentialoperators explizit bekannt sein muss, um die Integralgleichungen aufstellen zu können. Dies schränkt die Verwendbarkeit der BEM auf einige wenige, aber praktisch sehr relevante lineare Differentialgleichungen ein. Ein weiterer Nachteil ist, dass zum Auswerten der auftretenden schwach singulären Integrale aufwendige Quadraturverfahren implementiert werden müssen. Geeignete Methoden wurden in der Literatur bereits eingehend studiert, vgl. Sauter/Schwab 2004 [25]. Außerdem sind die auftretenden Matrizen im Gegensatz zum Fall der Finiten Elemente voll besetzt, sodass hier zusätzliche Kompressionstechniken angewendet werden müssen, um den Speicheraufwand zu verringern. Nennenswert sind die schnelle Multipolmethode (Greengard/Rokhlin 1997 [17]), die Methode des Panelclustering (Hackbusch/Nowak 1989 [22]), hierarchische Matrizen (Hackbusch 1999 [21]) sowie die Waveletdiskretisierung (Dahmen/Schneider 1999 [10]). Motivation für diese Arbeit war die Tatsache, dass a posteriori Fehlerschätzung bzw. Adaptivität für die BEM im Gegensatz zur FEM noch nicht so gut verstanden ist.

1.1 Modellproblem

Sei $\Omega \subseteq \mathbb{R}^3$ ein Gebiet mit hinreichend glattem Rand $\Gamma := \partial\Omega$. Der Rand $\Gamma = \Gamma_D \cup \Gamma_N$ sei disjunkt zerlegt in den Dirichletrand Γ_D und den Neumannrand Γ_N. Wir betrachten die Laplacegleichung mit gemischten Randbedingungen

$$\begin{aligned} -\Delta u &= f_L \quad \text{in} \quad \Omega, \\ u &= g_D \quad \text{auf} \quad \Gamma_D, \\ \partial_n u &= g_N \quad \text{auf} \quad \Gamma_N. \end{aligned}$$

Die Fundamentallösung des Laplaceoperators ist durch

$$G(z) = \frac{1}{4\pi\|z\|}$$

gegeben (Steinbach 2003 [27, Kapitel 5.1]). Wir definieren die drei Integraloperatoren

$$\widetilde{N} f_L(x) := \int_\Omega G(x-y) f_L(y)\, dy \quad \text{für} \quad x \in \Omega,$$

$$\widetilde{V}\phi(x) := \int_\Gamma G(x-y)\phi(y)\, ds_y \quad \text{für} \quad x \in \Omega$$

und

$$\widetilde{K} v(x) := \int_\Gamma \partial_{n(y)} G(x-y) v(y)\, ds_y \quad \text{für} \quad x \in \Omega.$$

Wir bezeichnen mit $\gamma_0 : H^1(\Omega) \to H^{1/2}(\Gamma)$ den Spuroperator und mit $\gamma_1 : H^1(\Omega) \to H^{-1/2}(\Gamma)$ die Konormalenableitung. In der schwachen Formulierung lautet die Laplacegleichung

$$\begin{aligned} -\Delta u &= f_L \in \widetilde{H}^{-1}(\Omega), \\ \gamma_0 u &= g_D \in H^{1/2}(\Gamma_D), \\ \gamma_1 u &= g_N \in \widetilde{H}^{-1/2}(\Gamma_N), \end{aligned}$$

wobei die erste Gleichung über die erste Greensche Formel verstanden wird. Wir stellen die Abbildungseigenschaften der vorgestellten Integraloperatoren zusammen, wobei $\widetilde{\Gamma} \subseteq \Gamma$ eine Teilmenge des Randes bezeichnet, zum Beispiel $\widetilde{\Gamma} \in \{\Gamma_D, \Gamma_N\}$.

Satz 1.1 (Steinbach 2003 [27, Kapitel 6.1]).
$\widetilde{N}$ kann als Operator $\widetilde{N} \in L(\widetilde{H}^{-1}(\Omega); H^1(\Omega))$ aufgefasst werden, und es gilt

$$-\Delta(\widetilde{N} f_L) = f_L \quad \forall f_L \in \widetilde{H}^{-1}(\Omega).$$

Insbesondere sind Spur und Konormalenableitung als Operatoren

$$\gamma_0 \widetilde{N} \in L(\widetilde{H}^{-1}(\Omega); H^{1/2}(\Gamma)) \quad \textit{und} \quad \gamma_1 \widetilde{N} \in L(\widetilde{H}^{-1}(\Omega); H^{-1/2}(\Gamma))$$

wohldefiniert. ■

Satz 1.2 (Steinbach 2003 [27, Kapitel 6.2, Kapitel 6.3]).
$\widetilde{V}$ kann als Operator $\widetilde{V} \in L(H^{-1/2}(\Gamma); H^1(\Omega))$ aufgefasst werden, und es gilt

$$-\Delta \widetilde{V}\phi = 0 \in \widetilde{H}^{-1}(\Omega) \quad \forall \phi \in H^{-1/2}(\Gamma).$$

Insbesondere sind Spur und Konormalenableitung als Operatoren

$$\gamma_0 \widetilde{V} \in L(\widetilde{H}^{-1/2}(\widetilde{\Gamma}); H^{1/2}(\widetilde{\Gamma})) \quad \textit{und} \quad \gamma_1 \widetilde{V} \in L(\widetilde{H}^{-1/2}(\widetilde{\Gamma}); \widetilde{H}^{-1/2}(\widetilde{\Gamma}))$$

wohldefiniert. ■

Satz 1.3 (Steinbach 2003 [27, Kapitel 6.4, Kapitel 6.5]).
$\widetilde{K}$ kann als Operator $\widetilde{K} \in L(H^{1/2}(\Gamma); H^1(\Omega))$ aufgefasst werden, und es gilt

$$-\Delta \widetilde{K} v = 0 \in \widetilde{H}^{-1}(\Omega) \quad \forall v \in H^{1/2}(\Gamma).$$

Insbesondere sind Spur und Konormalenableitung als Operatoren

$$\gamma_0\widetilde{K} \in L(H^{1/2}(\widetilde{\Gamma}); H^{1/2}(\widetilde{\Gamma})) \quad \textit{und} \quad \gamma_1\widetilde{K} \in L(H^{1/2}(\widetilde{\Gamma}); \widetilde{H}^{-1/2}(\widetilde{\Gamma}))$$

wohldefiniert. ■

In diesem Operatorsetting lässt sich die Darstellungsformel

$$u = \widetilde{N} f_L + \widetilde{V}(\gamma_1 u) - \widetilde{K}(\gamma_0 u) \tag{1.1}$$

im schwachen Sinn wie in Sauter/Schwab 2004 [25, Satz 3.1.6] beweisen. Das bedeutet, dass die Lösung u der Differentialgleichung eindeutig bestimmt ist, sobald die Cauchydaten $(\gamma_0 u, \gamma_1 u) \in H^{1/2}(\Gamma) \times H^{-1/2}(\Gamma)$ bekannt sind. Wir definieren

- das Einfachschichtpotential $V := \gamma_0\widetilde{V}$,
- das Doppelschichtpotential $K := \gamma_0\widetilde{K} + \frac{1}{2}$,
- das adjungierte Doppelschichtpotential $K' := \gamma_1\widetilde{V} - \frac{1}{2}$,
- den hypersingulären Integraloperator $W := -\gamma_1\widetilde{K}$.

Indem man auf (1.1) den Spuroperator γ_0 und die Konormalenableitung γ_1 anwendet, erhält man das sogenannte Calderón-System

$$\begin{pmatrix}\gamma_0 u \\ \gamma_1 u\end{pmatrix} = \begin{pmatrix}\frac{1}{2} - K & V \\ W & \frac{1}{2} + K'\end{pmatrix}\begin{pmatrix}\gamma_0 u \\ \gamma_1 u\end{pmatrix} + \begin{pmatrix}\gamma_0\widetilde{N} f_L \\ \gamma_1\widetilde{N} f_L\end{pmatrix}.$$

Das Lösen der Symmschen Integralgleichung

$$V\phi = f \quad \text{auf} \quad \Gamma_D \tag{1.2}$$

mit Lösung $\phi = \gamma_1 u$ und gewisser rechter Seite $f \in H^{1/2}(\Gamma_D)$ ist somit ein Teilproblem beim Lösen der Laplacegleichung mit gemischten Randdaten (vgl. Steinbach 2003 [27, Kapitel 7.3]). Sie ist im Fall $\Gamma_D = \Gamma$ äquivalent zum Dirichletproblem und dient dieser Arbeit als Modellproblem. Die Analysis sowie die numerischen Experimente werden für das Einfachschichtpotential der Laplacegleichung durchgeführt.

1.2 Motivation und Kernresultate

Die a posteriori Analysis für die Randelementmethode ist weitaus weniger entwickelt als für die Methode der Finiten Elemente. Eine nahezu vollständige Übersicht ist in Carstensen/Faermann 2001 [6] zu finden. Die Energienorm ist im Fall der Randelementmethode nicht lokal, sodass Strategien entwickelt werden müssen, um a posteriori Fehlerschätzer zu gewinnen, die lokale Information bereitstellen. Die Beiträge solcher Fehlerschätzer können dann als Indikatoren für einen adaptiven Netzverfeinerungsalgorithmus verwendet werden, der auch im Falle singulärer Lösungen mit optimaler Ordnung konvergiert.

In der Literatur wurden unter anderem zum Beispiel in Faermann 1998 [14] und in Carstensen/Stephan 1995 [9] residuale Fehlerschätzer vorgeschlagen und analysiert. Das Residuum ist in der Symmschen Integralgleichung von der Gestalt

$$R = f - V\phi$$

mit

$$f = (K + 1/2)g.$$

Insbesondere ist die punktweise Auswertung von R in der Regel nur approximativ möglich und zugleich rechenintensiv. Darüberhinaus können Integrale über das Residuum im Allgemeinen nur durch numerische Quadraturverfahren und nicht exakt berechnet werden. Die von Faermann 1998 [14] vorgeschlagene Methode zur Fehlerschätzung bedingt außerdem die Berechnung einer Sobolev-Slobodeckij Seminorm, deren Implementierung in praxi schwierig ist. In Steinbach 2000 [26] wird ein anderer Zugang zur Fehlerschätzung gewählt. Die auftretenden Terme zur Berechnung des Fehlerschätzers können ebenfalls nur durch numerische Verfahren und nicht exakt berechnet werden.

In der vorliegenden Arbeit werden mit Hilfe von Lokalisierungstechniken aus Carstensen/Praetorius 2006 [8] einfache Fehlerschätzer entwickelt, die exakt berechenbar sind. Die Bestimmung der involvierten Größen benötigt nämlich im Wesentlichen die Galerkin-Matrix sowie die Berechnung gewichteter L^2-Normen für stückweise polynomiale Ansatzfunktionen. Der sogenannte $h-h/2$-Schätzer, den wir eingehend analysieren, wird in dieser Arbeit erstmals für die Randelementmethode vorgestellt.

Für die Fehlerschätzer aus Faermann 1998 [14] und Steinbach 2000 [26] wurde Zuverlässigkeit und Effizienz bewiesen, wobei die Konstanten nicht bestimmt wurden. Auch der residuale Schätzer aus Carstensen/Stephan 1995 [9] ist zuverlässig mit unbekannter Konstante, die Effizienz wurde in Carstensen 1996 [5] aber lediglich für quasiuniforme Gitter und zweidimensionale Probleme bewiesen.

Die in dieser Arbeit entwickelten Fehlerschätzer η_H und η_M sind effizient mit bekannter Konstante $C_{\text{eff}} = 1$. Unter der Saturationsannahme

$$|||\phi - \phi_{h/2}|||/|||\phi - \phi_h||| \leq q < 1$$

sind sie auch zuverlässig mit Konstante $C_{\text{rel}} = 1/\sqrt{1-q^2}$. Dabei bezeichnen ϕ_h und $\phi_{h/2}$ die über einem Netz $\mathcal{T}_h$ bzw. dessen uniformer Verfeinerung $\mathcal{T}_{h/2}$ berechneten Galerkinlösungen. Experimentell beobachten wir für gewisse anisotrope Netzfolgen die asymptotische Exaktheit der Fehlerschätzer.

Die analytischen Methoden beruhen auf der Lokalisierung der Norm in $\widetilde{H}^{-1/2}(\Gamma)$ sowie der Elliptizität des Integraloperators. Daher gelten die erzielten Resultate offensichtlich auch für Probleme der linearen Elastostatik (Steinbach 2003 [27, Kapitel 1.2]) sowie für das Stokes-Problem (Steinbach 2003 [27, Kapitel 1.3]).

1.3 Aufbau der Arbeit

Der mathematische Hintergrund der Galerkin-Randelementmethode einschließlich der Definition der beteiligten Funktionenräume wird in Kapitel 2 wiederholt. Wir erklären den Begriff der Fastregularität einer Triangulierung, der in Folge in die Analysis als Voraussetzung einfließt. Wenn h_T den Durchmesser und ϱ_T den maximalen Inkreisradius eines Elementes $T \in \mathcal{T}$ einer Triangulierung bezeichnen, so definieren wir die Formregularität

$$\sigma(\mathcal{T}) = \max_{T\in\mathcal{T}} \frac{h_T}{\varrho_T}.$$

Diese wichtige Kenngröße fließt in diverse Abschätzungen ein, sodass einige Ergebnisse nur für isotrope Netzfolgen, die durch die Beschränktheit von $\sigma(\mathcal{T})$ charakterisiert sind, gezeigt werden. Die Beschaffenheit der Ansatzräume sowie die Konvergenz und eine a priori Fehlerschätzung für die Galerkin-Randelementmethode werden rekapituliert. Wir zeigen, dass eine Konvergenzrate von $\mathcal{O}(h^{3/2})$ erreicht werden kann.

Im Zentrum dieser Arbeit steht Kapitel 3. Dort werden einige $h-h/2$ und glättungsbasierte a posteriori Fehlerschätzer definiert und eingehend studiert. Wir betrachten zwei Triangulierungen $\mathcal{T}_h$ und

$\mathcal{T}_{h/2}$, wobei letztere aus $\mathcal{T}_h$ durch uniforme Verfeinerung hervorgeht. Wir bezeichnen mit ϕ_h und $\phi_h^{(1)}$ die diskreten Galerkinlösungen über dem Netz $\mathcal{T}_h$ für stückweise konstante bzw. stückweise lineare Ansatzfunktionen. Entsprechend ist $\phi_{h/2}$ die diskrete Galerkinlösung über dem feineren Netz $\mathcal{T}_{h/2}$ für stückweise konstante Ansatzfunktionen. Mit $\mathbb{G}_h$, $\mathbb{G}_{h/2}$ bzw. $\mathbb{G}_h^{(1)}$ werden die Galerkinprojektionen auf die jeweiligen Räume bezeichnet, mit Π_h, $\Pi_{h/2}$ und $\Pi_h^{(1)}$ die entsprechenden L^2-Projektionen. Mit dieser Notation definieren wir die Fehlerschätzer

$$\eta_H = |\!|\!| \phi_{h/2} - \phi_h |\!|\!| \quad \text{und} \quad \eta_M = |\!|\!| \phi_{h/2} - \mathbb{G}_h^{(1)} \phi_{h/2} |\!|\!|.$$

Wir beweisen

$$\eta_H \leq |\!|\!| \phi - \phi_h |\!|\!| \leq C_1 \eta_H \quad \text{und} \quad C_2 \eta_M \leq |\!|\!| \phi - \phi_{h/2} |\!|\!| \leq C_3 \eta_M,$$

wobei die Konstanten $C_1, C_2, C_3 > 0$ nicht von der Größe oder der Anzahl der Elemente abhängen. Die Zuverlässigkeit und Effizienz von η_M wurde erstmals unter starken Voraussetzungen in Carstensen/Praetorius 2006 [8] bewiesen. Wie bereits oben erwähnt, ist die Energienorm $|\!|\!| \cdot |\!|\!|$ nicht lokal. Das heißt, die Norm einer Funktion kann nicht als Summe lokaler Beiträge

$$|\!|\!| u |\!|\!|^2 \neq C \sum_{T \in \mathcal{T}} |\!|\!| u|_T |\!|\!|^2$$

geschrieben werden. Daher werden Lokalisierungstechniken benötigt, um Verfeinerungsindikatoren zu gewinnen. Wir beweisen eine Approximationseigenschaft für die L^2-Projektion und zitieren eine inverse Abschätzung aus Graham/Hackbusch/Sauter 2005 [19], die die Fastregularität des zugrunde liegenden Netzes voraussetzt. Mithilfe der Approximationseigenschaft und der inversen Abschätzung lässt sich die Äquivalenz der Fehlerschätzer

$$\widetilde{\mu}_H = \|\varrho^{1/2}(\phi_{h/2} - \Pi_h \phi_{h/2})\|_{L^2(\Gamma)} \sim \eta_H,$$
$$\widetilde{\mu}_M = \|\varrho^{1/2}(\phi_{h/2} - \Pi_h^{(1)} \phi_{h/2})\|_{L^2(\Gamma)} \sim \eta_M$$

bis auf die Formregularität zeigen. Dabei heißen zwei Größen a und b äquivalent $a \sim b$, falls es Konstanten $C_4, C_5 > 0$ gibt, sodass $C_4 a \leq b \leq C_5 a$ gilt. Abschließend beweisen wir für Rechtecksnetze die Äquivalenz $\eta_H \sim \eta_M$, wenn man η_M als Schätzer für den Fehler $|\!|\!| \phi - \phi_h |\!|\!|$ auf dem groben Netz auffasst. Dadurch können die Voraussetzungen für die Zuverlässigkeit und Effizienz von η_M im Vergleich zum Ergebnis aus Carstensen/Praetorius 2006 [8] wesentlich abgeschwächt werden. Wie bereits anfangs erwähnt, ist ein unmittelbarer Vorteil der vier vorgeschlagenen Fehlerschätzer, ist dass die Implementierung quasi keinen Zusatzaufwand erfordert und alle Größen exakt berechnet werden können.

In Kapitel 4 wird die in C++ implementierte Programmbibliothek FLBEM vorgestellt, die eigens für die Durchführung numerischer Experimente im Rahmen dieser Arbeit entstanden ist. Die Netzverwaltung wird ausführlich erklärt, wobei auf die Problematik der Erhaltung der Fastregularität eingegangen wird. Die Netzverwaltung hat linearen Aufwand und verbraucht im Vergleich zum Lösen des Gleichungssystems kaum Systemressourcen. Die Bibliothek bietet die Möglichkeit die Systemmatrizen approximativ als $\mathcal{H}$-Matrizen aufzubauen. Dem zugrunde liegt die Verwendung der Bibliothek *HLib*, die vom Max-Planck Institut für Mathematik in den Naturwissenschaften entwickelt und zur Verfügung gestellt wurde. Zur analytischen Berechnung der Matrixkoeffizienten wird die Routine *lapinteg3* aus dem Fortran-Programmpaket *maiprogs* verwendet, die die Stammfunktionen aus Maischak 2000 [23] implementiert.

Abschließend werden numerische Experimente in Kapitel 5 durchgeführt, die die Zuverlässigkeit und Effizienz der eingeführten Fehlerschätzer bestätigen. Wir beobachten, dass die Formregularität keinen Einfluss auf die Güte der Schätzer hat. Das stützt die Hypothese, dass die Approximationsresultate aus Kapitel 3 und damit die Zuverlässigkeit der Schätzer auch für gewisse anisotrope Netze gelten.

1.4 Danksagung

Ich möchte mich bei vielen Menschen dezidiert bedanken. Zunächst bei meinem Betreuer Dirk Praetorius, der jederzeit bereit war, auf kompetente Weise meine Fragen zu beantworten und mich in die richtige Richtung zu lenken. Weiters bin ich meiner geliebten Frau Bianca Ferraz-Leite zu Dank verpflichtet, die mich in allen Belangen des Lebens tatkräftig unterstützt und so manche schlaflose Nacht beim Korrekturlesen verbracht hat. Meine Eltern Elzbieta und Heber Ferraz-Leite müssen hier besondere Erwähnung finden, nicht nur für das Geschenk des Lebens und einer glücklichen Kindheit, sondern darüber hinaus weil sie mich stets in meinem Bestreben nach Wissen motiviert haben. Isabella Roth und Stephan Krenn hatten immer ein offenes Ohr, wenn ich einen Mathematiker zum Diskutieren benötigte. Johannes Kropf gilt mein Dank für die technische Unterstützung. Stefan Funken und Matthias Maischak haben mir Teile ihrer Quelltexte zur Verfügung gestellt und mich bei der Implementierung beraten. Dafür nochmals ein großes Dankeschön. Abschließend möchte ich Hans Kaiser erwähnen, ohne dessen Vorlesung “Algebra für Informatiker” ich nie herausgefunden hätte, dass meine wahre Leidenschaft der Mathematik gilt.

Wien, am 16.10.2007

Kapitel 2

Die Randelementmethode

In diesem Kapitel führen wir die Analysis für die Galerkin-Randelementmethode durch. Die a priori Fehlerschätzung wird für die Entwicklung der Fehlerschätzer in Kapitel 3 eine tragende Rolle spielen. Im Wesentlichen werden aus der Literatur bekannte Ergebnisse zusammengetragen. Wir halten uns dabei an die Arbeiten Praetorius 2007 [24] sowie die Monographien Sauter/Schwab 2004 [25] und Steinbach 2003 [27]. Eine möglichst einheitliche Darstellung wird angestrebt, daher werden in einem einführenden Abschnitt grundlegende Begriffe aus der Analysis der partiellen Differentialgleichungen erklärt. Ebenfalls werden die fundamentalen Sätze aus der Theorie der Sobolevräume zusammengetragen, sofern sie in Folge benötigt werden. Die Analysis der Randelementmethode wird nur für den Spzialfall der Symmschen Integralgleichung durchgeführt, sodass einige der zitierten Sätze auf diesen Fall reduziert bzw. adaptiert werden. Es war notwendig, manche Beweise zu vervollständigen bzw. zu korrigieren.

Um die Effizienz und die Zuverlässigkeit der Fehlerschätzer in Kapitel 3 zu gewährleisten, müssen schwache Forderungen an die Triangulierung gestellt werden. Die Begriffe der K-Eigenschaft sowie die geeignete Sichtweise für fastreguläre Netze sind Ferraz-Leite/Praetorius 2007 [16] entnommen.

2.1 Analytische Grundlagen

In diesem Abschnitt werden Sobolevräume auf Gebieten und Rändern definiert. Das sind die geeigneten Funktionenräume um partielle Differentialgleichungen zu studieren. Weiters werden einige Sätze, die später in die Analysis einfließen hier ebenfalls zitiert.

2.1.1 Sobolevräume

Es bezeichne $\Omega \subseteq \mathbb{R}^3$ ein orientierbares Lipschitzgebiet. Mit $\Gamma \subseteq \partial\Omega$ wird ein Teil des Randes des Gebietes bezeichnet. Wir führen Sobolevräume auf Gebieten ein.

Definition. Eine Funktion $u \in L^1_{\mathrm{loc}}(\Omega)$ heißt schwach differenzierbar, falls es eine Funktion $\partial_j u \in L^1_{\mathrm{loc}}(\Omega)$ gibt, sodass die Formel der partiellen Integration

$$\int_\Omega u(\partial_j v)\, dx = -\int_\Omega (\partial_j u) v\, dx \quad \forall v \in \mathcal{C}_0^\infty(\Omega)$$

erfüllt ist.

Definition. Der Sobolevraum $H^0(\Omega)$ wird mit $L^2(\Omega)$ identifiziert. Weiters definiert man

$$H^1(\Omega) := \{u \in L^2(\Omega) \mid u \text{ ist schwach differenzierbar mit } \nabla u \in L^2(\Omega)^d\}.$$

Die zugehörige Norm wird durch $\|u\|^2_{H^1(\Omega)} := \|u\|^2_{L^2(\Omega)} + \|\nabla u\|^2_{L^2(\Omega)}$ gegeben. Sobolevräume höherer ganzzahliger Ordnung werden induktiv durch

$$H^k(\Omega) := \{u \in L^2(\Omega) \mid u \text{ ist schwach differenzierbar mit } \nabla u \in H^{k-1}(\Omega)\}$$

mit der zugehörigen Norm $\|u\|^2_{H^k(\Omega)} := \|u\|^2_{L^2(\Omega)} + \|\nabla u\|^2_{H^{k-1}(\Omega)}$ definiert. Auf $H^k(\Omega)$ ist

$$|u|_{H^k(\Omega)} := \|D^k u\|_{L^2(\Omega)}$$

mit $D^k u$ der k-ten schwachen Ableitung von u eine Seminorm.

Definition. Für reelle Zahlen $0 < s < 1$ führt man zunächst die Sobolev-Slobodeckij Seminorm

$$|u|_{s,\Omega} := \Big(\int_\Omega \int_\Omega \frac{|u(x) - u(y)|^2}{|x-y|^{d+2s}} \Big)^{1/2}$$

ein. Mit $k \in \mathbb{N}_0$ und $0 < s < 1$ kann man mit Hilfe dieser Seminorm Sobolevräume mit nicht ganzzahliger Ordnung durch

$$H^{k+s} := \{u \in H^k(\Omega) \mid |D^k u|_{s,\Omega} < \infty\}$$

mit der Norm $\|u\|^2_{H^{k+s}(\Omega)} := \|u\|^2_{H^k(\Omega)} + |D^k u|^2_{s,\Omega}$ definieren.

Satz 2.1 (Evans 1998 [13, Abs. 5.2 Theorem 2]). *Für $s \geq 0$ ist $H^s(\Omega)$ ein Hilbertraum.* ■

Um die Definition des Dualraumes für Sobolevräume in der gängigen Darstellung über das erweiterte L^2-Skalarprodukt rechtfertigen zu können, soll an dieser Stelle noch ein allgemeines Lemma bewiesen werden.

Lemma 2.2. *Seien X, Y reelle Hilberträume mit stetiger Einbettung $X \subseteq Y$. Dann ist die Rieszabbildung*

$$J_Y : Y \to Y^*, \quad J_Y y := (y; \cdot)_Y$$

als Operator $J_y \in L(Y, X^*)$ wohldefiniert und $J_Y(Y)$ ist ein dichter Unterraum von X^*.

Beweis. Laut Voraussetzung gibt es eine Konstante $C > 0$, sodass $\|x\|_Y \leq C\|x\|_X$ für beliebiges $x \in X$. Aus der Cauchyungleichung folgt

$$(y; x)_Y \leq \|y\|_Y \|x\|_Y \leq C\|y\|_Y \|x\|_X.$$

Somit ist $J_y \in L(Y, X^*)$ wohldefiniert. Sei J_X die Rieszabbildung für X. Dann ist $J_Y(Y)$ genau dann dicht in X^*, wenn $V := J_X^{-1}(J_Y(Y))$ dicht in $X = \bar{V} \oplus \bar{V}^\perp$ ist. Es bleibt also lediglich $\bar{V}^\perp = \{0\}$ zu zeigen. Für $x \in \bar{V}^\perp$ und $y \in Y$, gilt

$$0 = (x; J_X^{-1}(J_Y(y)))_X = (J_Y(y))(x) = (y; x)_Y.$$

Mit der zulässigen Wahl $y = x \in \bar{V}^\perp$ folgt $x = 0$ in $Y \supseteq X$. ■

Man beachte, dass das allgemein formulierte Lemma auf den Fall der Sobolevräume mit $X = H^s(\Omega)$ und $Y = L^2(\Omega)$ angewandt werden kann. Insbesondere kann $L^2(\Omega)$ als dichter Teilraum von $H^s(\Omega)^*$ aufgefasst werden und die duale Klammer $\langle f, v\rangle$ für $v \in H^s(\Omega)$ und $f \in H^s(\Omega)^*$ stimmt mit dem L^2-Skalarprodukt überein, sofern $f \in L^2(\Omega)$ gilt. Daher können Sobolevräume negativer Ordnung

folgendermaßen definiert werden:

Definition. Mit $\widetilde{H}^{-s}(\Omega)$ für $s \geq 0$ wird der bezüglich des erweiterten L^2-Skalarproduktes dargestellte Dualraum von $H^s(\Omega)$ bezeichnet. Diese Räume heißen Sobolevräume der Ordnung $-s$.

Sobolevräume auf dem Rand eines Gebietes werden analog definiert. Für eine genaue Charakterisierung benötigen wir noch den Begriff der Hölderstetigkeit.

Definition. Eine Funktion $f : \Omega \to \mathbb{R}$ heißt hölderstetig mit Ordnung (k, λ), falls $f \in \mathcal{C}^k(\Omega)$ ist und alle k-ten Ableitungen von f mit Multiindizes α und $x, y \in \Omega$

$$\sup_{|\alpha|=k} \sup_{x \neq y} \frac{|\partial^\alpha f(x) - \partial^\alpha f(y)|}{|x-y|^\lambda} < \infty$$

erfüllen. Der Raum aller hölderstetigen Funktionen der Ordnung (k, λ) wird mit $\mathcal{C}^{k,\lambda}$ bezeichnet.

Definition. Ein Gebiet Ω heißt $\mathcal{C}^{k,\lambda}$-Gebiet, falls es ein Lipschitzgebiet ist, dessen lokalen Parametrisierungen hölderstetig mit Ordnung (k, λ) sind.

Satz 2.3 (Wloka 1964 [28, Satz 4.2]). *Sei Ω ein $\mathcal{C}^{k-1,k}$-Gebiet und $0 \leq s \leq k$. Dann ist der Sobolevraum $H^s(\Gamma)$ für beliebige Parametrisierungen von Γ als Menge eindeutig bestimmt. Verschiedene Parametrisierungen führen zu äquivalenten Normen. Insbesondere gilt $L^2(\Gamma) = H^0(\Gamma)$ mit äquivalenter Norm. $H^s(\Gamma)$ ist ein Hilbertraum und die Einbettung $H^t(\Gamma) \subseteq H^s(\Gamma)$ für $s \leq t \leq k$ ist stetig.* ■

Lemma 2.2 kann mit $X = H^s(\Gamma)$ und $Y = L^2(\Gamma)$ angewendet werden. Daher kann man auch auf Rändern die Dualräume analog zum Fall der Sobolevräume auf Gebieten definieren.

Definition. Die Sobolevräume negativer Ordnung werden als Dualraum $\widetilde{H}^{-s}(\Gamma) = H^s(\Gamma)^*$ bezüglich des erweiterten L^2-Skalarproduktes definiert.

Im Folgenden werden wir uns hauptsächlich für die Sobolevräume $H^1(\Omega)$ und $H^{1/2}(\Gamma)$ interessieren. Der folgende wichtige Satz stellt eine direkte Beziehung zwischen diesen Räumen her und erklärt, wie man den Randraum aus dem Gebietsraum gewinnen kann.

Satz 2.4 (Evans 1998 [13, Abs. 5.5 Theorem 1]). *Es gibt einen eindeutigen Spuroperator $\gamma_0 \in L(H^1(\Omega); H^{1/2}(\Gamma))$, sodass $\gamma_0 u = u|_\Gamma$ für alle $u \in \mathcal{C}^\infty(\bar{\Omega})$ gilt. Das bedeutet, dass $\gamma_0 u$ die Einschränkung einer Sobolevfunktion $u \in H^1(\Omega)$ auf Γ darstellt.* ■

2.1.2 Sätze aus der Analysis

Bevor wir uns den Ergebnissen aus der Theorie der Sobolevräume widmen, wollen wir noch einen Satz aus der Funktionalanalysis beweisen, der in der Theorie der Galerkin-Randelementmethode benötigt wird.

Satz 2.5. *Sei $A \in L(X, Y)$ ein kompakter Operator und sei $(x_n) \rightharpoonup x$ eine schwach konvergente Folge in X. Dann ist das Bild $(Ax_n) \to Ax$ stark konvergent in Y.*

Beweis. Offensichtlich ist $(Ax_n) \rightharpoonup Ax$ schwach konvergent. Angenommen Ax_n konvergiert nicht im starken Sinne, dann gibt es eine Teilfolge Ax_{n_k} mit $\inf_{k\in\mathbb{N}}\|Ax_{n_k} - Ax\|_Y \geq \varepsilon$ für ein gewisses $\varepsilon > 0$. Da A ein kompakter Operator ist, folgt die Existenz einer weiteren Teilfolge $(Ax_{n_{k_l}})$, sodass $Ax_{n_{k_l}} \to y \in Y$ gilt. Insbesondere konvergiert diese Folge auch schwach gegen y womit aus der Eindeutigkeit des Grenzwertes sofort $y = Ax$ folgt, was ein Widerspruch zur postulierten Teilfolge (Ax_{n_k}) ist. ■

Wir zitieren einige wichtige Eigenschaften von Sobolevräumen.

Satz 2.6 (Meyers-Serrin. Evans 1998 [13, Abs. 5.3 Theorem 2]). *Für beliebiges $s \geq 0$ ist $\mathcal{C}^\infty(\Omega) \cap H^s(\Omega)$ ein dichter Unterraum von $H^s(\Omega)$. Insbesondere ist $\mathcal{C}^\infty(\bar{\Omega}) = \{u|_\Omega$ mit $u \in \mathcal{C}^\infty(\mathbb{R}^d)\}$ ein dichter Unterraum des Sobolevraums $H^s(\Omega)$.* ■

Satz 2.7 (Sobolevungleichung. Evans 1998 [13, Abs. 5.6 Theorem 6]). *Für $\frac{d}{2} < s$ gilt $H^s(\Omega) \subseteq \mathcal{C}(\Omega)$ stetiger Einbettung. Das heißt $\|u\|_\infty \lesssim \|u\|_{H^s(\Omega)}$ für beliebiges $u \in H^s(\Omega)$.* ■

Satz 2.8 (Kompaktheitssatz von Rellich. Alt 2002 [1, Satz 8.9]). *Für beliebiges $s > t$ ist die Einbettung $H^s(\Omega) \subseteq H^t(\Omega)$ kompakt. Das heißt die Identität ist ein kompakter Operator.*

Satz 2.9 (Poincaré-Ungleichung. Evans 1998 [13, Abs. 5.8 Theorem 1]). *Für alle $u \in H^1(\Omega)$ gilt*

$$\|u\|_{L^2(\Omega)} \leq C_p \mathrm{diam}(\Omega)\Big(\|\nabla u\|_{L^2(\Omega)} + \Big|\int_\Omega u \, dx\Big|\Big). \tag{2.1}$$

Insbesondere erhält man die Abschätzung $\|u\|_{L^2(\Omega)} \leq C_p \mathrm{diam}(\Omega)\|\nabla u\|_{L^2(\Omega)}$, falls $\int_\Omega u \, dx = 0$ ist. Im Falle konvexer Gebiete Ω kann gezeigt werden, dass die Ungleichung mit $C_p = \frac{1}{\pi}$ hält. ■

Wir wollen uns nun mit der komplexen Interpolation von Banachräumen befassen, die ein mächtiges Werkzeug darstellt. Dabei beschränken wir uns auf den hier interessanten Fall der Interpolation von ineinander stetig eingebetteten separablen Banachräumen $X_1 \subseteq X_0$. Wir vereinfachen die Ausführungen in Bergh/Löfström 1976 [2] entsprechend. Man betrachtet den Funktionenraum $\Phi(X_1, X_0)$, der alle Funktionen $\phi : \mathbb{C} \to X_0$ enthält, die mit $M := \{z|0 \leq \mathcal{R}(z) \leq 1\}$

(i) $\phi(z)$ ist stetig und beschränkt auf M

(ii) $\phi(z)$ ist analytisch auf M

(iii) $\phi(it) \in X_1$ für $t \in \mathbb{R}$ und $\phi(it)$ ist analytisch bezüglich X_1

erfüllen. Der Interpolationsraum $X_\alpha = [X_1; X_0]_\alpha$ mit $0 \leq \alpha \leq 1$ besteht aus allen Elementen $x \in X_0$, die sich durch $x = \phi(\alpha)$ mit $\phi \in \Phi(X_1, X_0)$ darstellen lassen. Auf X_α ist durch

$$\|x\|_{X_\alpha} = \inf_{\phi(\alpha)=x} \|\phi\|_{\Phi(X_1,X_0)}$$

eine Norm gegeben, mit der der Interpolationsraum ein Banachraum ist. Die zugehörige Norm erfüllt

$$\|\cdot\|_{X_\alpha} \leq \|\cdot\|_{X_0}^{1-\alpha}\|\cdot\|_{X_1}^{\alpha}.$$

Satz 2.10 (Interpolationssatz. BERGH/LÖFSTRÖM 1976 [2, Teorem 4.1.2]). *Seien $X_1 \subseteq X_0$ und $Y_1 \subseteq Y_0$ stetig eingebettete separable Banachräume. Der Operator T erfülle $T \in L(X_0; Y_0)$ und $T \in L(X_1; Y_1)$. Dann kann T auch als Operator $T \in L(X_\alpha; Y_\alpha)$ auf den Interpolationsräumen mit Operatornorm*

$$\|T\|_{L(X_\alpha;Y_\alpha)} \leq \|T\|^{1-\alpha}_{L(X_0;Y_0)} \|T\|^{\alpha}_{L(X_1;Y_1)} \tag{2.2}$$

aufgefasst werden.

Bemerkung. Aus dem Interpolationssatz folgt sofort, dass der interpolierte Raum H_α zweier stetig eingebetteter separabler Hilberträume $H_1 \subseteq H_0$ selbst ein Hilbertraum ist. Durch Interpolation von $H^1(\Gamma) \subseteq H^0(\Gamma)$ gewinnt man eine alternative Definition der Sobolevräume $H^\alpha(\Gamma) = [H^1(\Gamma); H^0(\Gamma)]_\alpha$ mit äquivalenter Norm. Als Spezialfall wird der Interpolationssatz häufig auf Operatoren angewendet, die auf $H^0 = L^2$ bzw. H^1 operieren, um Aussagen über die Räume H^α zu gewinnen.

Als Beispiel für eine Anwendung des Interpolationssatzes wollen wir folgendes Lemma beweisen.

Lemma 2.11. *Sei $\mathcal{T} = \{\Gamma_1, \ldots, \Gamma_n\}$ eine Zerlegung des Randstückes Γ. Für $0 \leq \alpha \leq 1$ und $u \in H^\alpha(\Gamma)$ gilt*

$$\sum_{j=1}^{n} \|u|_{\Gamma_j}\|^2_{H^\alpha(\Gamma_j)} \leq \|u\|^2_{H^\alpha(\Gamma)}.$$

Beweis. Wir betrachten die Räume $H^\alpha(\mathcal{T})$ mit $0 \leq \alpha \leq 1$, die als Produkträume der $H^\alpha(\Gamma_j)$ definiert sind. Die zugehörige Norm ist durch $\|\cdot\|_{H^\alpha(\mathcal{T})} = \Big(\sum\limits_{j=1}^{n} \|\cdot\|^2_{H^\alpha(\Gamma_j)}\Big)^{1/2}$ gegeben. Offensichtlich sind die $H^\alpha(\mathcal{T})$ separable Hilberträume. Wegen

$$\|u\|^2_{L^2(\Gamma)} = \sum_{j=1}^{n} \|u\|^2_{L^2(\Gamma)}$$

und

$$\|u\|^2_{H^1(\Gamma)} = \|u\|^2_{L^2(\Gamma)} + \|\nabla u\|^2_{L^2(\Gamma)} = \sum_{j=1}^{n} (\|u\|^2_{L^2(\Gamma_j)} + \|\nabla u\|^2_{L^2(\Gamma_j)}) = \sum_{j=1}^{n} \|u|_{\Gamma_j}\|^2_{H^1(\Gamma_j)}$$

gelten die Identifikationen $H^0(\Gamma) = H^0(\mathcal{T})$ und $H^1(\Gamma) = H^1(\mathcal{T})$. Die Identität $\mathbb{I} \in L(H^0(\Gamma); H^0(\mathcal{T}))$, $\mathbb{I} \in L(H^1(\Gamma); H^1(\mathcal{T}))$ hat jeweils Norm $\|\mathbb{I}\| = 1$. Laut Interpolationssatz kann die Identität als Operator $\mathbb{I} \in L(H^\alpha(\Gamma); H^\alpha(\mathcal{T}))$ mit Norm $\|\mathbb{I}\| \leq 1$ aufgefasst werden. Daher gilt

$$\|u\|_{H^\alpha(\mathcal{T})} = \|\mathbb{I}u\|_{H1\alpha(\mathcal{T})} \leq \|\mathbb{I}\| \|u\|_{H^\alpha(\Gamma)} \leq \|u\|_{H^\alpha(\Gamma)}.$$

Man beachte, dass aus dem Beweis insbesondere hervorgeht, dass für $u \in H^\alpha(\Gamma)$ die Einschränkung $u|_{\Gamma_j}$ ein Element des Raumes $H^\alpha(\Gamma_j)$ ist. Wir haben die Tatsache, dass der Interpolationsraum von Produkträumen mit dem Produktraum von Interpolationsräumen ident ist, benutzt. ■

2.2 Variationsformulierung des Modellproblems

In diesem Abschnitt begründen wir auf einfache Weise die eindeutige Lösbarkeit der Variationsformulierung der Symmschen Integralgleichung. Wir gehen dabei wie in SAUTER/SCHWAB 2004 [25,

Kapitel 4.1.1] vor, wobei die Argumentation auf das Anwenden des Darstellungssatzes von Riesz reduziert wird.

Die Gleichung

$$V\phi = f \tag{2.3}$$

im Funktionenraum $H^{1/2}(\Gamma)$ ist nach Satz von Hahn-Banach äquivalent zur Variationsformulierung

$$\langle V\phi, \psi\rangle = \langle f, \psi\rangle \quad \forall \psi \in \widetilde{H}^{-1/2}(\Gamma), \tag{2.4}$$

da $\widetilde{H}^{-1/2}(\Gamma)$ der Dualraum von $H^{1/2}(\Gamma)$ ist. Nach Definition von $\widetilde{H}^{-1/2}(\Gamma)$ stimmt dabei die duale Klammer für $\psi \in L^2(\Gamma)$ mit dem L^2-Skalarprodukt überein, das heißt es gilt

$$\int_\Gamma (V\phi)\psi ds_x = \int_\Gamma \psi(x) \int_\Gamma \frac{\phi(y)}{4\pi\|x-y\|} ds_y ds_x = \int_\Gamma f(x)\psi(x) d_x.$$

Für Funktionen $\phi, \psi \in L^\infty(\Gamma)$ gilt laut Satz von Fubini

$$\langle V\phi, \psi\rangle = \int_\Gamma \int_\Gamma \frac{\phi(x)\psi(y)}{4\pi\|x-y\|} ds_y ds_x = \langle V\psi, \phi\rangle,$$

daher ist V symmetrisch. Das Einfachschichtpotential $V : \widetilde{H}^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ ist stetig und elliptisch (Steinbach 2003, [27, Lemma 6.3 und Satz 6.5]), das heißt, es gibt eine Konstante $\gamma > 0$ mit

$$\gamma \|u\|^2_{\widetilde{H}^{-1/2}(\Gamma)} \leq \langle Vu, u\rangle \leq \|V\| \|u\|^2_{\widetilde{H}^{-1/2}(\Gamma)}.$$

Daher definiert $\langle\!\langle u, v\rangle\!\rangle = \langle Vu, v\rangle$ in (2.4) ein äquivalentes Skalarprodukt auf dem Energieraum $\widetilde{H}^{-1/2}(\Gamma)$ mit der induzierten Energienorm $|\!|\!| u |\!|\!| = \langle\!\langle u, u\rangle\!\rangle^{1/2}$. Die rechte Seite in (2.4) definiert mit $F(\psi) = \langle f, \psi\rangle$ wegen $f \in H^{1/2}(\Gamma) = \left(\widetilde{H}^{-1/2}(\Gamma)\right)^*$ ein stetiges lineares Funktional. Laut dem Darstellungssatz von Riesz existiert somit eine eindeutige Lösung des Variationsproblems.

Die Idee der Galerkin-Randelementmethode ist es, den Funktionenraum $\widetilde{H}^{-1/2}(\Gamma)$ für ϕ, ψ durch einen endlichdimensionalen geeigneten Raum zu ersetzen.

2.3 Triangulierung

Um geeignete diskrete Räume zu konstruieren, wollen wir zunächst das Konzept der Triangulierung des Randes Γ erklären.

Definition. Wir definieren die Referenzelemente

$$\widehat{T}_D = \{x \in \mathbb{R}^2 | x_1, x_2 > 0, x_1 + x_2 < 1\} \quad \text{und} \quad \widehat{T}_R = (0,1)^2$$

für Dreieckspaneele bzw. Rechteckspaneele.

Wir definieren eine Triangulierung zunächst sehr allgemein und lehnen uns dabei an Sauter/Schwab 2004 [25, Kapitel 4.1.2] an. Im Folgenden werden wir uns allerdings auf einen einfacheren Fall beschränken, sodass die abstrakt wirkenden Regularitätsbedingungen für χ_T stets erfüllt ist.

Definition. Eine Unterteilung des Randes in endlich viele relativ offene und disjunkte Teilstücke $T_1, \ldots, T_n \subseteq \Gamma$ heißt Triangulierung (bzw. Netz oder Paneelierung) $\mathcal{T}$, falls folgende Bedingungen erfüllt sind:

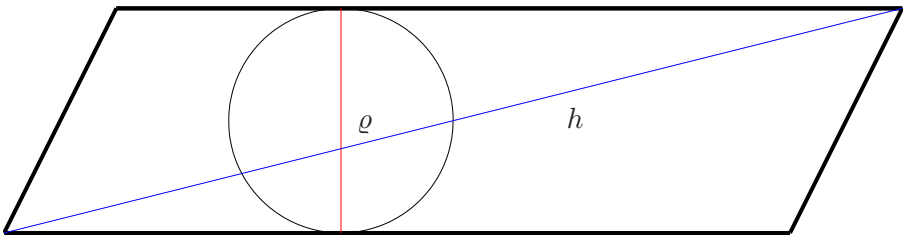


Abbildung 2.1: Kenngrößen des Netzes h, ϱ

(i) $\Gamma = \overline{\bigcup_{T\in\mathcal{T}} T}$.

(ii) Jedes Element T ist das Bild eines Referenzelementes $\widehat{T} \in \{\widehat{T}_D, \widehat{T}_R\}$ unter einer regulären Abbildung χ_T. Hier bedeutet regulär, dass χ_T mit der zugehörigen Jacobi-Matrix J_T die Bedingung

$$0 < \lambda_{\min} \leq \inf_{\widehat{x}\in\widehat{T}} \inf_{\substack{v\in\mathbb{R}^2\\ \|v\|=1}} \langle J_T(\widehat{x})v, J_T(\widehat{x})v\rangle \leq \sup_{\widehat{x}\in\widehat{T}} \sup_{\substack{v\in\mathbb{R}^2\\ \|v\|=1}} \langle J_T(\widehat{x})v, J_T(\widehat{x})v\rangle \leq \lambda_{\max} < \infty$$

erfüllt.

Um den für die Analysis der a posteriori Fehlerschätzer in Kapitel 3 richtigen Begriff einer Triangulierung zu erhalten definieren wir zunächst reguläre Netze. Anschließend wird die Regularität stückweise verallgemeinert. Wir folgen den Ideen in Ferraz-Leite/Praetorius 2007 [16, Abschnitt 2].

Definition. Eine Triangulierung heißt regulär, falls:

(i) Der Schnitt des Abschlusses zweier verschiedener Elemente $T, T' \in \mathcal{T}$ entweder leer, ein gemeinsamer Punkt oder eine gemeinsame Kante ist.

(ii) Liegt im Schnitt eine gemeinsame Kante $e = \overline{T} \cap \overline{T'}$ vor, stimmen die Fortsetzungen der Parametrisierungen entlang der Kante überein. D.h.

$$\chi_T|_{\widehat{e}} = \chi_{T'} \circ \gamma_{T,T'}|_{\widehat{e}}$$

ist mit einer geeigneten affinen Bijektion $\gamma_{T,T'} : \widehat{T} \to \widehat{T}$ erfüllt.

Bemerkung. Die Notation und Analysis im allgemeinsten Fall ist aufwendig, gewährt aber keine didaktisch tieferen Einblicke. Wir wollen uns für diese Abhandlung auf den Fall einer Polyederoberfläche Γ beschränken, die sich vollständig durch ebene Dreiecke oder ebene Rechtecke überdecken lässt. In diesem Fall sind die Abbildungen χ_T der zugehörigen Triangulierung affin und erfüllen die zweite Bedingung aus der Definition, sofern kein entartetes Dreieck bzw. Rechteck vorliegt.

Definition. Für jedes Element $T_j \in \mathcal{T}$ einer Paneelierung definieren wir den Durchmesser $h_j := \operatorname{diam}(T_j) > 0$. Mit $\varrho_j > 0$ bezeichnen wir den Durchmesser des größten eingeschriebenen Inkreises in T_j. Weiters definieren wir die lokalen Netzweiten $h, \varrho \in L^\infty$ durch $h|_{T_j} = h_j$ bzw. $\varrho|_{T_j} = \varrho_j$. Die lokalen Netzweiten sind in Abbildung 2.1 illustriert.

Bemerkung. Da der Durchmesser bezüglich einer beliebigen Norm verstanden werden kann, ist es möglich und meist einfacher bei einer Implementierung nicht die euklidische sondern die Maximumnorm zu verwenden, sodass der Durchmesser rechteckiger Elemente der Betrag einer längsten Seite ist.

Definition. Wir definieren die globale Maschenweite eines Netzes durch

$$h_{\max}(\mathcal{T}) := \max\{h|_\tau \text{ mit } \tau \in \mathcal{T}\}.$$

Definition. Wir definieren die Konstante der Formregularität eines Netzes durch

$$\sigma(\mathcal{T}) := \sup_{T_j \in \mathcal{T}} (h_j/\varrho_j) = \|h/\varrho\|_{L^\infty(\Gamma)} \geq 1.$$

Da wir sukzessive diskrete Näherungen der Lösung auf immer feiner werdenden Netzen berechnen wollen, wird es interessant sein, die Entwicklung der vorgestellten Kenngrößen zu betrachten.

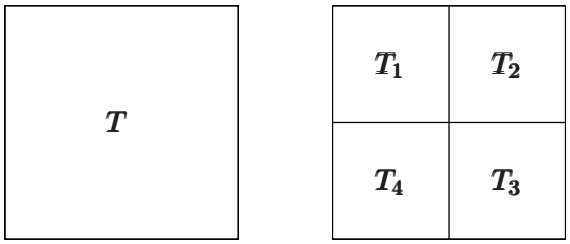


Abbildung 2.2: Isotrope Verfeinerung eines Rechtecks T, um vier neue Rechtecke $T_1, \ldots, T_4$ mit gleicher Formregularität h/ϱ zu erhalten.

Definition. Ein Element $T \in \mathcal{T}$ wird isotrop in vier Elemente $T_1, \ldots, T_n$ geteilt, wenn $h_{T_i} = h_T/2$ und $\varrho_{T_i} = \varrho_T/2$ sind. Dies kann durch Teilung aller Kanten wie in Abbildung 2.2 erreicht werden. Wir sagen, ein Netz $\mathcal{T}_{h/2}$ geht durch uniforme Verfeinerung aus einem Netz $\mathcal{T}_h$ hervor, wenn jedes Element in $\mathcal{T}_{h/2}$ aus einem Element von $\mathcal{T}_h$ durch isotrope Verfeinerung hervorgeht.

Definition. Eine Folge von Netzen $(\mathcal{T}_\ell)_{\ell \in \mathbb{N}}$ heißt isotrop, falls $\sigma := \sup_{\ell \in \mathbb{N}} \sigma(\mathcal{T}_\ell) < \infty$ gilt. Ansonsten heißt die Folge anisotrop.

Aus numerischen Experimenten weiß man, dass isotrope Verfeinerung nicht in allen Fällen zur optimalen Konvergenzrate führt. Es ist notwendig, Anisotropie zuzulassen, um Kantensingularitäten effektiv zu approximieren. Daher werden wir schwächere Bedingungen an die Triangulierungen stellen müssen. Die Fastregularität und die K-Eigenschaft, beides schwache Forderungen an das Netz, erweisen sich für die Analysis der Fehlerschätzer als hinreichend.

Definition. Ein Netz erfüllt die K-Eigenschaft, falls es eine Konstante $\kappa(\mathcal{T}) \geq 1$ gibt, sodass

(i) für $\overline{T_j} \cap \overline{T_k} \neq \emptyset$ gelten $h_j/h_k \leq \kappa(\mathcal{T})$ und $\varrho_j/\varrho_k \leq \kappa(\mathcal{T})$

(ii) für jeden Knoten z der Triangulierung gilt $|\{T \in \mathcal{T} | z \in \overline{T}\}| \leq \kappa(\mathcal{T})$

erfüllt sind.

Da wir in den numerischen Experimenten Rechtecksnetze betrachten wollen, müssen wir eine der Regularitätsforderungen, nämlich, dass jeder Knoten ein Eckpunkt ist, fallen lassen, da sonst keine lokalen Verfeinerungen möglich sind. Wir führen fastreguläre Netze ein.

Definition. Wenn $z \in \overline{T_1} \cap \overline{T_2}$ ein Eckpunkt von T_1 aber nicht von T_2 ist, so heißt z hängender Knoten, siehe Abbildung 2.3.

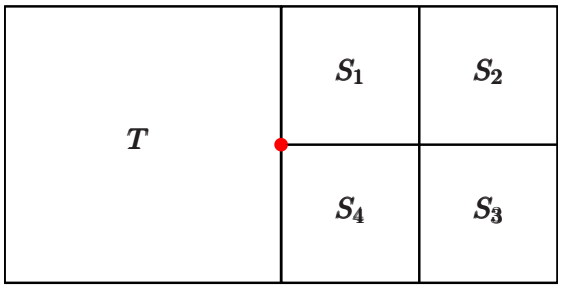


Abbildung 2.3: Ein hängender Knoten, im Bild rot eingezeichnet, ist Eckpunkt eines Elementes S_1 bzw. S_4, liegt aber auf der Kante des benachbarten Rechtecks T.

Definition. Eine Triangulierung $\mathcal{T}$ heißt fastregulär, falls der nichtleere Schnitt des Abschlusses zweier Elemente entweder die identisch parametrisierte Kante eines der Elemente, ein Eckpunkt beider Elemente oder ein hängender Knoten ist. Weiter wird die Existenz eines Netzes $\widetilde{\mathcal{T}}$ gefordert, sodass gelten:

(i) $\widetilde{\mathcal{T}}$ geht durch Verfeinerungen aus $\mathcal{T}$ hervor. Insbesondere ist jedes Element $\widetilde{T} \in \widetilde{\mathcal{T}}$ Teilmenge genau eines Elementes $T \in \mathcal{T}$.

(ii) Es gibt eine Konstante $\widetilde{\kappa}(\mathcal{T})$, die nur von $\mathcal{T}$ abhängt, sodass

$$\kappa(\widetilde{\mathcal{T}}) \leq \widetilde{\kappa}(\mathcal{T})\,\kappa(\mathcal{T}) \tag{2.5}$$

und

$$\widetilde{h} \leq h \leq \widetilde{\kappa}(\mathcal{T})\,\widetilde{h} \quad \text{sowie} \quad \widetilde{\varrho} \leq \varrho \leq \widetilde{\kappa}(\mathcal{T})\,\varrho. \tag{2.6}$$

Dabei bezeichnen h und ϱ sowie $\widetilde{h}$ und $\widetilde{\varrho}$ die entsprechenden Netzkenngrößen von $\mathcal{T}$ und $\widetilde{\mathcal{T}}$.

In praxi werden wir fastreguläre Netze mit der K-Eigenschaft erhalten, indem wir nur einen hängenden Knoten auf jeder Kante zulassen und darauf achten, dass das Verhältnis der lokalen Netzweiten benachbarter Elemente beschränkt bleibt.

Bemerkung. Das Verbot hängender Knoten in der üblichen Definition regulärer Netze rührt daher, dass oft mit global stetigen Ansatzfunktionen gearbeitet wird. Wir werden uns auf unstetige Randelemente konzentrieren, wodurch die entsprechende Regularitätsforderung fallen gelassen werden kann.

Definition. Für eine Triangulierung $\mathcal{T} = \{T_1, \ldots, T_n\}$ definieren wir für $\alpha \geq 0$ die Sobolevräume über der Triangulierung als Produktraum

$$H^\alpha(\mathcal{T}) = \prod_{j=1}^{n} H^\alpha(T_j)$$

mit der Norm

$$\|u\|^2_{H^\alpha(\mathcal{T})} = \sum_{j=1}^{n} \|u|_{T_j}\|^2_{H^\alpha(T_j)}.$$

Die Sobolevräume negativer Ordnung über einer Triangulierung werden analog definiert.

Bemerkung. Laut Lemma 2.11 ist die Einschränkung jeder Funktion $u \in H^\alpha(\mathcal{T})$ auf ein Paneel $u|_{T_j}$ ein Element des Raumes $H^\alpha(T_j)$. Insbesondere kann $H^\alpha(\Gamma) \subseteq H^\alpha(\mathcal{T})$ als entsprechender Teilraum aufgefasst werden.

2.4 Unstetige Randelemente

Zur Approximation der unbekannten Lösung ϕ der Symmschen Integralgleichung soll ein endlichdimensionaler Ansatz gefunden werden. Bei der Galerkin-Randelementmethode erreicht man dieses Ziel, indem man als Ansatzfunktionen in der Variationsformulierung nur stückweise Polynome auf einem Netz $\mathcal{T}$ zulässt. Da die Lösung der Symmschen Integralgleichung in $\widetilde{H}^{-1/2}(\Gamma)$ liegt, müssen unstetige Ansatzfunktionen sinnvollerweise zugelassen werden. Als Beispiel wollen wir wie in Sauter/Schwab 2004 [25, Kapitel 4.1.3] den Fall der niedrigst möglichen Ordnung betrachten, nämlich die stückweise konstanten Funktionen.

Definition. Für eine fastreguläre Triangulierung $\mathcal{T}$ wird der Raum der stückweise konstanten Funktionen durch

$$\mathcal{P}^0(\mathcal{T}) := \{\psi \in L^\infty(\Gamma) \,|\, \forall T \in \mathcal{T} : \psi|_T \text{ ist konstant}\}$$

definiert. Jede Funktion $\psi \in \mathcal{P}^0(\mathcal{T})$ ist eindeutig durch ihre Werte ψ_T auf den Randelementen definiert. Somit bilden die charakteristischen Funktionen der Randelemente eine offensichtlich endliche Basis von $\mathcal{P}^0(\mathcal{T})$.

Um Randelemente höherer Ordnung definieren zu können, müssen wir zunächst die Polynome von partiellem Grad p auf einem Referenzelement erklären.

Definition. Wir definieren den Raum der Polynome von partiellem Grad p auf einem Referenzelement $\widehat{T}$ durch

$$\widehat{\mathcal{P}}^p = \operatorname{span}\{\widehat{x}^\mu \,|\, \mu \in \mathbb{N}^2 \text{ und } |\mu| \leq p\}.$$

Hierbei bezeichnet μ einen Multiindex. In Folge definieren wir den Raum der stückweisen Polynome von partiellem Grad p über einem Netz $\mathcal{T}$ durch

$$\mathcal{P}^p(\mathcal{T}) = \{\psi : \Gamma \to \mathbb{R} \,|\, \forall T \in \mathcal{T} : \psi \circ \chi_T \in \widehat{\mathcal{P}}^p\}.$$

Bemerkung. Man beachte, dass laut Definition der Raum $\mathcal{P}^p(\mathcal{T})$ insbesondere von der gewählten Parametrisierung abhängig ist, da die Ansatzfunktionen auf jedem Element erst nach Rücktransport auf das Referenzelement Polynome sind. Da wir uns jedoch auf den Fall von stückweise konstanten Funktionen beschränken, sind die Ansatzfunktionen unabhängig von der Parametrisierung eindeutig bestimmt. Man beachte, dass man im Falle höherer Polynomordnung die Parametrisierungen bei Netzverfeinerungen sinnvoll wählen kann, sodass die Räume entsprechend der Ordnung und der Feinheit des Netzes geschachtelt sind.

2.5 Die Galerkin-Randelementmethode

Wir betrachten im Sinne einer möglichst einfachen Darstellung nur den Fall stückweise konstanter Ansatzfunktionen und erklären die Galerkin-Randelementmethode für die Symmsche Integralgleichung in Anlehnung an Sauter/Schwab 2004 [25, Kapitel 4.1.4].

Setzt man in die Variationsformulierung des Modellproblems (2.4) statt $\phi \in \widetilde{H}^{-1/2}(\Gamma)$ eine endlichdimensionale Ansatzfunktion $\phi_h \in \mathcal{P}^0(\mathcal{T}_h)$ ein, so entsteht das Problem, dass die Gleichheit im Allgemeinen nicht erfüllt sein kann. Man muss zusätzlich den Raum der Testfunktionen diskretisieren. Sinnvollerweise ersetzen wir auch ψ durch ein diskretes $\psi_h \in \mathcal{P}^0(\mathcal{T}_h)$. Die Galerkin-Approximation

der Integralgleichung (1.2) lautet also:
Finde $\phi_h \in \mathcal{P}^0(\mathcal{T}_h)$, sodass

$$\langle\!\langle \phi_h, \psi_h \rangle\!\rangle = \langle f, \psi_h \rangle \quad \forall \psi_h \in \mathcal{P}^0(\mathcal{T}_h)$$

gilt.

Bemerkung. Das Ermitteln einer Approximation durch die Galerkin-Randelementmethode entspricht genau dem Lösen eines Gleichungssystems

$$B\varphi = F \tag{2.7}$$

mit $\varphi \in \mathbb{R}^n$. Hierbei ist B eine Matrix mit Einträgen

$$B_{i,j} = \int_\Gamma \int_\Gamma \frac{b_i(x) b_j(y)}{4\pi \|x-y\|} ds_y ds_x$$

wobei die Menge $\{b_1, \ldots, b_n\}$ eine Basis von $\mathcal{P}^0(\mathcal{T}_h)$ bildet. Üblicherweise wählt man b_i als charakteristische Funktion eines Elementes $T_i \in \mathcal{T}_h$. Die rechte Seite ist ein Vektor F mit $F_i = \int_\Gamma f(x) b_i(x) ds_x$. Man beachte, dass die Matrix B voll besetzt ist. In praxi wird sie jedoch nur approximativ in Form einer $\mathcal{H}$-Matrix gespeichert, siehe Börm/Grasedyck/Hackbusch 2003 [4].

Satz 2.12. *Die Systemmatrix B in (2.7) ist symmetrisch und positiv definit.*

Beweis. Aus der Symmetrie des Skalarproduktes $\langle\!\langle \phi, \psi \rangle\!\rangle = \langle\!\langle \psi, \phi \rangle\!\rangle$ folgt sofort

$$B_{i,j} = \langle\!\langle b_i, b_j \rangle\!\rangle = \langle\!\langle b_j, b_i \rangle\!\rangle = B_{j,i}.$$

Sei nun $\varphi \in \mathbb{R}^n$ beliebig und $\phi_h = \sum_{j=1}^n \varphi_j b_j$. Dann gilt

$$\begin{aligned} \varphi^T B \varphi &= \sum_{i,j} \varphi_j \varphi_i B_{i,j} = \sum_{i,j} \varphi_j \varphi_i \langle\!\langle b_i, b_j \rangle\!\rangle = \langle\!\langle \sum_i \varphi_i b_i, \sum_j \varphi_j b_j \rangle\!\rangle \\ &= \langle\!\langle \phi_h, \phi_h \rangle\!\rangle \geq \gamma \|\phi_h\|^2_{\widetilde{H}^{-1/2}(\Gamma)} > 0 \end{aligned}$$

genau dann, wenn $\phi_h \neq 0$ also genau denn, wenn $\varphi \neq 0 \in \mathbb{R}^n$. ■

Das diskrete Problem besitzt eine eindeutige Lösung, die mit geeigneten iterativen Verfahren, zum Beispiel mit der CG-Methode, gefunden werden kann. Wir wollen uns nun Gedanken über den Verfahrensfehler machen.

Der folgende Satz von Céa wird wie in Sauter/Schwab 2004 [25, Proposition 4.1.24] bewiesen, wobei wir die Formulierung in der Energienorm in den Satz einfließen lassen.

Satz 2.13 (Céa). *Sei ϕ die analytische Lösung der Symmschen Integralgleichung (1.2). Die Galerkinlösung ϕ_h konvergiert Optimal in der Energienorm*

$$|\!|\!| \phi - \phi_h |\!|\!| = \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} |\!|\!| \phi - \psi_h |\!|\!|. \tag{2.8}$$

In der $\widetilde{H}^{-1/2}(\Gamma)$-Norm folgt mit γ der Elliptizitätskonstante von V die Quasioptimalität

$$\|\phi - \phi_h\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \Big(\frac{\|V\|}{\gamma}\Big)^{1/2} \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} \|\phi - \psi_h\|_{\widetilde{H}^{-1/2}(\Gamma)}. \tag{2.9}$$

Der Fehler erfüllt die Galerkinorthogonalität

$$\langle\!\langle \phi - \phi_h, \psi_h \rangle\!\rangle = 0 \quad \forall \psi_h \in \mathcal{P}^0(\mathcal{T}_h). \tag{2.10}$$

Beweis. Wir zeigen zunächst die Galerkinorthogonalität. Man beachte, dass sowohl die analytisch exakte Lösung als auch die Galerkinlösung $\langle\!\langle \phi, \psi_h \rangle\!\rangle = \langle f, \psi_h \rangle$ und $\langle\!\langle \phi_h, \psi_h \rangle\!\rangle = \langle f, \psi_h \rangle$ erfüllen. Daher gilt

$$\langle\!\langle \phi - \phi_h, \psi_h \rangle\!\rangle = \langle\!\langle \phi, \psi_h \rangle\!\rangle - \langle\!\langle \phi_h, \psi_h \rangle\!\rangle = \langle f, \psi_h \rangle - \langle f, \psi_h \rangle = 0.$$

Unter Ausnutzung der Galerkinorthogonalität erhält man für den Fehler $e_h = \phi - \phi_h$ die Abschätzung

$$\begin{aligned} |\!|\!| e_h |\!|\!|^2 &= \langle\!\langle e_h, e_h \rangle\!\rangle = \langle\!\langle e_h, \phi - \phi_h \rangle\!\rangle \\ &= \langle\!\langle e_h, \phi \rangle\!\rangle - \langle\!\langle e_h, \phi_h \rangle\!\rangle = \langle\!\langle e_h, \phi \rangle\!\rangle - \langle\!\langle e_h, \psi_h \rangle\!\rangle = \langle\!\langle e_h, \phi - \psi_h \rangle\!\rangle \\ &\leq |\!|\!| e_h |\!|\!| \, |\!|\!| \phi - \psi_h |\!|\!| \end{aligned}$$

für beliebiges $\psi_h \in \mathcal{P}^0(\mathcal{T}_h)$. Minimierung liefert

$$|\!|\!| \phi - \phi_h |\!|\!| \leq \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} |\!|\!| \phi - \psi_h |\!|\!|,$$

damit folgt wegen $\phi_h \in \mathcal{P}^0(\mathcal{T}_h)$ die Gleichung (2.8). Die Aussage (2.9) erhält man unmittelbar aus der Normäquivalenz

$$\begin{aligned} \gamma \|e_h\|^2_{\widetilde{H}^{-1/2}(\Gamma)} \leq |\!|\!| e_h |\!|\!|^2 = \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} |\!|\!| \phi - \psi_h |\!|\!|^2 &= \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} \langle V(\phi - \psi_h), \phi - \psi_h \rangle \\ &\leq \min_{\psi_h \in \mathcal{P}^0(\mathcal{T}_h)} \|V\| \, \|\phi - \psi_h\|^2_{\widetilde{H}^{-1/2}(\Gamma)}. \end{aligned}$$

Nach Division der Ungleichung durch γ und Ziehen der Wurzel ist der Beweis vollständig. ■

Korollar 2.14. *Das Galerkinverfahren definiert für einen diskreten Raum X_h einen Lösungsoperator $\mathbb{G} : \widetilde{H}^{-1/2}(\Gamma) \to X_h$. Diese sogenannte Galerkinprojektion $\mathbb{G}$ ist die Orthogonalprojektion auf X_h in der Energienorm.* ■

Der folgende Satz zeigt die Konvergenz des Galerkinverfahrens für beliebige durch Verfeinerungen auseinander hervorgehende Netze. Die Netzfolgen müssen nicht fastregulär sein. Die Formregularität fließt nicht in den Beweis ein. In Sauter/Schwab 2004 [25, Korollar 4.1.27] wird der Satz für beliebige Netzfolgen $\mathcal{T}_\ell$ mit $h_\ell \to 0$ formuliert. Die Argumentation scheint jedoch zu fordern, dass alle Netze geschachtelt sind. Das heißt $T \in \mathcal{T}_\ell$ muss sich als Vereinigung von Elementen in $\mathcal{T}_{\ell-1}$ darstellen lassen. Sonst ist nicht klar, dass die Treppenfunktionen $\mathcal{P}^0(\mathcal{T}_\ell)$ tatsächlich den Raum $L^2(\Gamma)$ approximieren.

Satz 2.15. *Sei $(\mathcal{T}_\ell)_{\ell \in \mathbb{N}}$ eine Folge von geschachtelten Triangulierungen mit Netzweiten $h_\ell \to 0$. Jedes Element aus $\mathcal{T}_\ell$ lasse sich als Vereinigung von Elementen in $\mathcal{T}_{\ell-1}$ darstellen. Dann konvergiert die Folge der Galerkinlösungen $(\phi_\ell)_{\ell \in \mathbb{N}}$ gegen die exakte Lösung ϕ.*

Beweis. Der Raum $\mathcal{P}^0(\mathcal{T}_\ell)$ besteht aus den Treppenfunktionen auf einem Gitter $\mathcal{T}_\ell$. Laut Definition der Lebesgue-Räume und $h_\ell \to 0$ ist also $\bigcup_{\ell \in \mathbb{N}} \mathcal{P}^0(\mathcal{T}_\ell)$ dicht in $L^2(\Gamma)$, welcher wiederum dicht in $\widetilde{H}^{-1/2}(\Gamma)$ ist. Für beliebiges $\varepsilon > 0$ lassen sich also $\widetilde{\phi} \in L^2(\Gamma)$, $\ell \in \mathbb{N}$ und $\widetilde{\phi}_\ell \in \mathcal{P}^0(\mathcal{T}_\ell)$ wählen, sodass

$$\|\phi - \widetilde{\phi}\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \frac{\varepsilon}{2}, \qquad \|\widetilde{\phi} - \widetilde{\phi}_\ell\|_{L^2(\Gamma)} \leq \frac{\varepsilon}{2}$$

gelten. Insgesamt ergibt sich

$$\|\phi - \widetilde{\phi}_\ell\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \|\phi - \widetilde{\phi}\|_{\widetilde{H}^{-1/2}(\Gamma)} + \|\widetilde{\phi} - \widetilde{\phi}_\ell\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \varepsilon.$$

Aus der Quasioptimalität folgt mit

$$\|\phi - \phi_\ell\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \Big(\frac{\|V\|}{\gamma}\Big)^{1/2} \|\phi - \widetilde{\phi}_\ell\|_{\widetilde{H}^{-1/2}(\Gamma)} \leq \varepsilon \Big(\frac{\|V\|}{\gamma}\Big)^{1/2}$$

die Konvergenz für $\ell \to \infty$. ■

2.6 Einfluss von Störungen

Bis jetzt haben wir uns noch nicht mit der für die Numerik zentralen Frage des Einflusses von Störungen befasst. Es stellt sich heraus, dass das Galerkinverfahren stabil bezüglich Störungen ist, sofern diese hinreichend klein bleiben. Wir führen die Beweise wie in Praetorius 2007 [24], wobei einige kleinere Ungenauigkeiten berichtigt wurden.

Lemma 2.16 (Strang). *Seien H ein Hilbertraum, $a(\cdot,\cdot)$ eine stetige, elliptische Bilinearform auf H, $\ell \in H^*$ und $u \in H$ mit $a(u,\cdot) = \ell \in H^*$. Für einen Diskretisierungsparameter h seien $X_h \leq H$ abgeschlossene Unterräume, $a_h(\cdot,\cdot)$ eine stetige Bilinearformen auf X_h und $\ell_h \in X_h^*$ ein gegebenes lineares Funktional. Es gelten folgende Aussagen:*

(i) Falls $\lim\limits_{h\to 0} \sup\limits_{v_h, w_h \in X_h \setminus \{0\}} \frac{|a(v_h,w_h) - a_h(v_h,w_h)|}{\|v_h\|_H \|w_h\|_H} = 0$ *gilt, so folgt*

$$\exists h_0 > 0 \exists \widetilde{\alpha} > 0 \forall h < h_0 \forall v_h \in X_h : \widetilde{\alpha}\|v_h\|_H^2 \leq a_h(v_h, v_h). \tag{2.11}$$

Insbesondere ist die Form $a_h(\cdot,\cdot)$ elliptisch auf X_h und es existiert somit ein eindeutiges $u_h \in X_h$, sodass $a_h(u_h,\cdot) = \ell_h \in X_h^*$ gilt.

(ii) Es gelte (2.11), so folgt

$$\|u - u_h\|_H \lesssim \inf_{v_h \in X_h} (\|u - v_h\|_H + \|a(v_h,\cdot) - a_h(v_h,\cdot)\|_{X_h^*}) + \|\ell - \ell_h\|_{X_h^*}.$$

Beweis.

(i) Gemäß der Annahme können wir ein $\varepsilon > 0$ und $h_0 > 0$ finden, sodass für alle $h < h_0$ die Ungleichung

$$\sup_{v_h \in X_h \setminus \{0\}} \frac{|a(v_h,v_h) - a_h(v_h,v_h)|}{\|v_h\|_H^2} \leq \varepsilon$$

erfüllt ist. Mit der Elliptizitätskonstante $\alpha > 0$ der Bilinearform $a(\cdot,\cdot)$ schließt man

$$\begin{aligned} \alpha\|v_h\|_H^2 &\leq a(v_h,v_h) = a_h(v_h,v_h) + a(v_h,v_h) - a_h(v_h,v_h) \\ &\leq a_h(v_h,v_h) + |a(v_h,v_h) - a_h(v_h,v_h)| \\ &\leq a_h(v_h,v_h) + \varepsilon\|v_h\|_H^2. \end{aligned}$$

Insgesamt erhält man die Abschätzung $(\alpha - \varepsilon)\|v_h\|_H^2 \leq a_h(v_h,v_h)$ und damit die Aussage mit $\widetilde{\alpha} = \alpha - \varepsilon$.

(ii) Für $v_h \in X_h$ gilt

$$\begin{aligned}
\|u_h - v_h\|_H^2 &\lesssim a_h(u_h - v_h, u_h - v_h) \\
&= \ell_h(u_h - v_h) - a_h(v_h, u_h - v_h) \\
&= \ell_h(u_h - v_h) - a_h(v_h, u_h - v_h) - a(u - v_h, u_h - v_h) + \\
&\quad a(v_h, u_h - v_h) - a(u, u_h - v_h) \\
&\lesssim \ell_h(u_h - v_h) - a_h(v_h, u_h - v_h) - \|u - v_h\|_H \|u_h - v_h\|_H + \\
&\quad a(v_h, u_h - v_h) - \ell(u_h - v_h) \\
&\lesssim (\ell_h(u_h - v_h) - \ell(u_h - v_h)) + \|u - v_h\|_H \|u_h - v_h\|_H + \\
&\quad (a(v_h, u_h - v_h) - a_h(v_h, u_h - v_h)) \\
&\lesssim (\|\ell_h - \ell\|_{X_h^*} + \|a(v_h, \cdot) - a_h(v_h, \cdot)\|_{X_h^*} + \|u - v_h\|_H)\|u_h - v_h\|_H.
\end{aligned}$$

Wegen $\|u - u_h\|_H \leq \|u - v_h\|_H + \|u_h - v_h\|_H$ folgt die Abschätzung für beliebiges v_h. Bilden des Infimums liefert schließlich die Behauptung.

■

Wir zeigen die Konvergenz der gestörten Galerkin-BEM in einem allgemeinen Rahmen, sodass man den Zusammenhang zum Lemma von Strang besser erkennt. Natürlich umfasst diese allgemeine Formulierung den Fall der Symmschen Integralgleichung.

Satz 2.17 (Konvergenz der gestörten Galerkin-BEM). *Seien H ein Hilbertraum, $a(\cdot,\cdot)$ eine stetige, elliptische Bilinearform, $\ell \in H^*$ ein lineares Funktional und $u \in H$, sodass $a(u,\cdot) = \ell \in H^*$. Für $h > 0$ seien weiters $X_h \leq H$ ein abgeschlossener Unterraum von H, $a_h(\cdot,\cdot)$ eine stetige Bilinearform auf X_h und $\ell_h \in X_h^*$ ein lineares Funktional, sodass*

$$\begin{aligned}
\lim_{h\to 0} \|\ell - \ell_h\|_{X_h^*} &= 0 \\
\lim_{h\to 0} \alpha_h &= 0 \quad \textit{mit} \quad \alpha_h := \sup_{v_h, w_h \in X_h \setminus \{0\}} \frac{|a(v_h, w_h) - a_h(v_h, w_h)|}{\|v_h\|_H \|w_h\|_H}
\end{aligned}$$

gelten. Weiters sei $u_h \in X_h$, sodass $a_h(u_h, \cdot) = \ell_h \in X_h^*$ gilt. Gibt es eine Dichte Teilmenge $D \subseteq H$, sodass

$$\lim_{h\to 0} \min_{v_h \in X_h} \|v - v_h\|_H = 0$$

gilt, so folgt

$$\lim_{h\to 0} \|u - u_h\|_H = 0.$$

Beweis. Sei $v \in D$. Mit dem Lemma von Strang und Anwenden der Dreiecksungleichung, sowie Einsetzen der Voraussetzungen gilt

$$\begin{aligned}
\|u - u_h\|_H &\lesssim \inf_{v_h \in X_h} (\|u - v_h\|_H + \|a(v_h, \cdot) - a_h(v_h, \cdot)\|_{X_h^*}) + \|\ell - \ell_h\|_{X_h^*} \\
&\leq \inf_{v_h \in X_h} (\|v - v_h\|_H + \|u - v\|_H + \alpha_h \|v_h\|_{X_h}) + \|\ell - \ell_h\|_{X_h^*} \\
&\leq \inf_{v_h \in X_h} (\|v - v_h\|_H + \|u - v\|_H + \alpha_h \|v - v_h\|_H + \alpha_h \|v\|_H) + \|\ell - \ell_h\|_{X_h^*} \\
&\leq \|u - v\|_H + \underbrace{(1 + \alpha_h)}_{\to 1} \underbrace{\inf_{v_h \in X_h} \|v - v_h\|_H}_{\to 0} + \underbrace{\alpha_h}_{\to 0} \|v\|_H + \underbrace{\|\ell - \ell_h\|_{X_h^*}}_{\to 0}.
\end{aligned}$$

Da $v \in D$ beliebig war und D eine dichte Teilmenge ist folgt die Aussage. ■

Bemerkung. Man beachte, dass der zuletzt formulierte Satz besagt, dass die gestörte Galerkin Randelementmethode jedenfalls konvergent ist, sofern die Störungen konsistent sind. In Praxis muss darauf geachtet werden, dass der Einfluss der Störungen in gleichem Maße verschwindet, wie der Verfahrensfehler.

2.7 A priori Fehlerschätzung

Wir haben gesehen, dass die Galerkin-Randelementmethode für $h \to 0$ konvergiert, aber noch nicht untersucht mit welcher Ordnung $\mathcal{O}(h^\alpha)$ das Verfahren konvergieren kann. Wir werden uns damit beschäftigen wie in Praetorius 2007 [24] Approximationsaussagen zu treffen, sodass auch diese Frage geklärt wird. Bis jetzt erfolgte die Analysis unter Einschränkung auf stückweise konstante Ansatzfunktionen. Für die Entwicklung der Fehlerschätzer benötigt man die Konvergenzaussage auch für Elemente höheren Grades. Daher werden wir die Ergebnisse in allgemeinerer Form erarbeiten.

Lemma 2.18 (Bramble-Hilbert). *Bezeichne $\mathcal{P}^k(\Omega)$ den Raum der Polynome von Grad kleiner oder gleich $k \in \mathbb{N}$ für ein Lipschitzgebiet Ω. Ist $f \in H^{k+1}(\Omega)^*$ ein stetiges lineares Funktional mit $ker(f) \supseteq \mathcal{P}^k(\Omega)$, so folgt*

$$|f(v)| \lesssim \|f\| |v|_{H^{k+1}(\Omega)}.$$

Insbesondere ist

$$\inf_{p \in \mathcal{P}^k(\Omega)} \|v - p\|_{H^{k+1}(\Omega)} \lesssim |v|_{H^{k+1}(\Omega)}.$$

Beweis. Wir halten zunächst fest, dass für die L^2-Projektion Π auf $\mathcal{P}^k(\Omega)$ jedenfalls

$$|f(v)| = |f(v - \Pi v)| \leq \|f\| \|v - \Pi v\|_{H^{k+1}(\Omega)}$$

gilt. Wir führen nun eine Norm $\|\cdot\|$ auf $H^{k+1}(\Omega)$ durch die Definition

$$\|w\| := |w|_{H^{k+1}(\Omega)} + \|\Pi w\|_{L^2(\Omega)}$$

ein. Da $\|w\| = 0$ impliziert, dass $w \in \mathcal{P}^k(\Omega)$ und $\Pi w = 0$ gilt, muss jedenfalls w identisch 0 sein. Daher ist klar, dass $\|\cdot\|$ definit ist. Überdies gilt $\|\cdot\| \lesssim \|\cdot\|_{H^{k+1}(\Omega)}$. Denn angenommen es wäre nicht so, dann existierte eine Folge derart, dass für jedes $n \in \mathbb{N}$ ein u_n gefunden werden kann, sodass

$$\frac{1}{n} \|u_n\|_{H^{k+1}(\Omega)} > |u_n|_{H^{k+1}(\Omega)} + \|\Pi u_n\|_{L^2(\Omega)}$$

gilt. Wir definieren $v_n := \frac{u_n}{\|u_n\|_{H^{k+1}(\Omega)}}$, dann gilt $\|v_n\|_{H^{k+1}(\Omega)} = 1$ für jedes $n \in \mathbb{N}$. Weiters ist

$$|v_n|_{H^{k+1}(\Omega)} + \|\Pi v_n\|_{L^2(\Omega)} \leq \frac{1}{n}.$$

Da $(v_n)_{n \in \mathbb{N}}$ eine beschränkte Folge in einem reflexiven Banachraum ist, gibt es eine schwach konvergente Teilfolge. O.B.d.A. sei $v_n \rightharpoonup v \in H^{k+1}(\Omega)$. Laut dem Satz von Rellich 2.8 ist die Einbettung $H^{k+1}(\Omega) \subseteq H^k(\Omega)$ kompakt. Da das kompakte Bild einer schwach konvergenten Folge laut Satz 2.5 stark konvergiert, gilt $v_n \to v \in H^k(\Omega)$. Die Seminorm $|v|_{H^{k+1}(\Omega)}$ verschwindet, daher muss $v \in \mathcal{P}^k(\Omega)$ sein. Insbesondere ist $v = \Pi v$. Da somit die Norm $\|v\|_{L^2(\Omega)}$ ebenfalls verschwindet, gilt insgesamt

$v \to 0 \in H^{k+1}(\Omega)$, was im Widerspruch zu $\|v_n\|_{H^{k+1}(\Omega)} = 1$ steht. Daher folgt $\|\cdot\| \lesssim \|\cdot\|_{H^{k+1}(\Omega)}$. Insgesamt haben wir durch

$$|f(v)| \leq \|f\|\|v - \Pi v\|_{H^{k+1}(\Omega)} \lesssim \|f\|\|v - \Pi v\| = \|f\| \underbrace{|v - \Pi v|_{H^{k+1}(\Omega)}}_{=|v|_{H^{k+1}(\Omega)}} + \underbrace{\|v - \Pi v\|_{L^2(\Omega)}}_{=0}$$

den ersten Teil des Lemmas bewiesen.

Um den zweiten Teil zu beweisen, bezeichne $\mathbb{P}$ die Orthogonalprojektion auf $\mathcal{P}^k(\Omega)$ bezüglich $\|\cdot\|_{H^{k+1}(\Omega)}$. Sei ϕ ein normiertes lineares Funktional, sodass

$$\|v - \mathbb{P}v\|_{H^k(\Omega)} = \phi(v - \mathbb{P}v) = \underbrace{\phi \circ (\mathbb{I} - \mathbb{P})}_{=:f} v$$

gilt. Das soeben definierte Funktional f erfüllt die Voraussetzungen um den ersten Teil des Satzes anzuwenden. Also gilt insgesamt

$$\inf_{p \in \mathcal{P}^k(\Omega)} \|v - p\|_{H^{k+1}(\Omega)} = \|v - \mathbb{P}v\|_{H^{k+1(\Omega)}} = |f(v)| \lesssim |v - \mathbb{P}v|_{H^{k+1}(\Omega)} = |v|_{H^{k+1}(\Omega)},$$

womit das Lemma vollständig bewiesen ist. ■

Der nun folgende Satz aus der Theorie der Sobolevräume soll nur zitiert und nicht bewiesen werden. Der Beweis ist nicht tiefgründig, sondern beschränkt sich im Wesentlichen auf sehr lange und technische Rechnungen.

Satz 2.19 (Transformationssatz. Sauter/Schwab 2004 [25, Lemma 4.3.6]). *Sei $\widehat{T}$ das Referenzelement zu einem Paneel T. Sei weiters χ_T die zugehörige Parametrisierung. Es ist $v \in H^k(\Omega)$ genau dann, wenn $\widehat{v} := v \circ \chi_T \in H^k(\Omega)$ ist. Weiters gelten*

$$\begin{aligned} |v|_{H^\ell(T)} &\lesssim h_T^{1-\ell} |\widehat{v}|_{H^\ell(\widehat{T})}, \\ |\widehat{v}|_{H^\ell(\widehat{T})} &\lesssim h_T^{\ell-1} |v|_{H^\ell(T)}. \end{aligned}$$

Die Konstanten in den Abschätzungen hängen nur vom Verhältnis $\frac{h_T^{d-1}}{|T|}$ ab, wobei $|\cdot|$ das Maß bezeichnet und d die Dimension des Raumes. ■

Lemma 2.20. *Für $0 \leq t$ und $\Pi : L^2 \to \mathcal{P}^p(\mathcal{T})$ der L^2-Projektion auf den Raum der stückweisen Polynome vom Grad kleiner oder gleich p über dem Netz $\mathcal{T}$ gilt*

$$\|u - \Pi u\|_{L^2(\Gamma)} \lesssim h_{\max}^{\min\{p+1,t\}} \|u\|_{H^t(\Gamma)}.$$

Beweis. Sei $u \in H^t(\Gamma)$. Wir betrachten zunächst den Fall $t \leq p+1, t \in \mathbb{N}$. Wir definieren ein lineares Funktional

$$g(\widehat{v}) := \int_{\widehat{T}} (\widehat{v} - \widehat{\Pi} v)(\zeta)(\widehat{u} - \widehat{\Pi} u)(\zeta) d\zeta.$$

Hierbei bezeichnet $\widehat{T}$ das Referenzelement und $\widehat{\Pi}$ die L^2-Projektion auf dem Referenzelement. Wegen

$$\begin{aligned} \|g\| = \sup_{\substack{\widehat{v} \in H^k(\widehat{T}) \\ \|v\|=1}} |g(\widehat{v})| &= \sup_{\substack{\widehat{v} \in H^k(\widehat{T}) \\ \|v\|=1}} \|\widehat{u} - \widehat{\Pi}\widehat{u}\|_{L^2(\widehat{T})} \|\widehat{v} - \Pi\widehat{v}\|_{L^2(\widehat{T})} \\ &\leq \sup_{\substack{\widehat{v} \in H^k(\widehat{T}) \\ \|v\|=1}} \|\widehat{u} - \Pi\widehat{u}\|_{L^2(\widehat{T})} \|\widehat{v}\|_{L^2(\widehat{T})} \\ &= \|\widehat{u} - \Pi\widehat{u}\|_{L^2(\widehat{T})} \end{aligned}$$

ist g ein stetiges Funktional. Offensichtlich ist außerdem $\ker(g) \supseteq \mathcal{P}^p(\widehat{T})$, also erfüllt g die Voraussetzungen um das Bramble-Hilbert Lemma 2.18 anwenden zu können. Insgesamt haben wir

$$|g(\widehat{v})| \lesssim \|\widehat{u} - \widehat{\Pi}\widehat{u}\|_{L^2(\widehat{T})} |\widehat{v}|_{H^t(\widehat{T})}.$$

Wir setzen nun $\widehat{v} = \widehat{u}$ und wenden den Transformationssatz 2.19 sowie ein einfaches Skalierungsargument für ein Element T an. Man beachte, dass $\widehat{u - \Pi u} = \widehat{u} - \widehat{\Pi}\widehat{u}$ gilt.

$$\begin{aligned}
\|u - \Pi u\|^2_{L^2(T)} &\lesssim h_T^2 \|\widehat{u} - \widehat{\Pi}\widehat{u}\|^2_{L^2(\widehat{T})} \\
&= h_T^2 |g(\widehat{u})| \lesssim h_T^2 \|\widehat{u} - \widehat{\Pi}u\|_{L^2(\widehat{T})} |\widehat{u}|_{H^t(\widehat{T})} \\
&\lesssim h_T^2 h_T^{-1} \|u - \Pi u\|_{L^2(T)} h_T^{t-1} |u|_{H^t(T)} \\
&= h_T^t \|u - \Pi u\|_{L^2(T)} |u|_{H^t(T)}
\end{aligned}$$

Summation über alle Elemente und Abschätzen von h_T durch $h_{\max}$ liefert die Behauptung

$$\|u - \Pi u\|_{L^2(\Gamma)} \lesssim h_{\max}^t \|u\|_{H^t(\Gamma)}.$$

Im Fall $t > p+1$ wird der Beweis analog mit der H^{p+1} Norm geführt. Diese kann durch die H^t Norm abschließend abgeschätzt werden. Für den Fall, dass t nicht ganzzahlig ist erhält man die Behauptung durch Interpolation des Operators

$$\mathbb{I} - \Pi \in L(L^2, L^2), L(H^t, L^2)$$

gemäß dem Interpolationssatz 2.10. ■

Satz 2.21. *Für $0 \leq s \leq t$ und Π der L^2-Projektion auf $\mathcal{P}^p(\mathcal{T})$ gilt*

$$\|u - \Pi u\|_{\widetilde{H}^{-s}(\Gamma)} \lesssim h_{\max}^{\min\{p+1,t\}+s} \|u\|_{H^t(\Gamma)}.$$

Beweis. Analog zum Beweis obigen Lemmas genügt es den Fall $t \leq p+1$ zu betrachten. Wir schätzen

$$\begin{aligned}
\|u - \Pi u\|_{\widetilde{H}^{-s}(\Gamma)} &= \sup_{v \in H^s(\Gamma)\setminus\{0\}} \frac{\langle u - \Pi u, v\rangle}{\|v\|_{H^s(\Gamma)}} \\
&= \sup_{v \in H^s(\Gamma)\setminus\{0\}} \frac{\langle u - \Pi u, v - \Pi v\rangle}{\|v\|_{H^s(\Gamma)}} \\
&\leq \underbrace{\|u - \Pi u\|_{L^2(\Gamma)}}_{\lesssim h_{\max}^t \|u\|_{H^t(\Gamma)}} \underbrace{\frac{\|v - \Pi v\|_{L^2(\Gamma)}}{\|v\|_{H^s(\Gamma)}}}_{\lesssim h_{\max}^s} \\
&\lesssim h_{\max}^{t+s} \|u\|_{H^t(\Gamma)}
\end{aligned}$$

ab, um den Satz zu beweisen. ■

Korollar 2.22. *Für $s \leq 0 \leq t, p \in \mathbb{N}$ gilt*

$$\inf_{v_h \in \mathcal{P}^p(\mathcal{T})} \|v - v_h\|_{\widetilde{H}^s(\Gamma)} \lesssim h_{\max}^{\min\{p+1,t\}-s} \|v\|_{H^t(\Gamma)}$$

für beliebiges $v \in H^t(\Gamma)$.

Wir haben gezeigt, dass die Galerkin-Randelementmethode für die Symmsche Integralgleichung mit unstetigen $\mathcal{P}^0(\mathcal{T})$ Ansatzelementen bei hinreichend glatter Lösung mit $\mathcal{O}(h^{-3/2})$ konvergiert. Bei der Konvergenzaussage ist allerdings die Formregularität über den Transformationssatz eingeflossen. In gewissem Sinne ist das suboptimal, da wir an sonst an keiner Stelle der Analysis des Galerkinverfahrens Regularität des Netzes fordern mussten. Mit Hilfe so genannter inverser Abschätzungen kann man zeigen, dass dies die optimale Konvergenzrate ist (Braess 1997 [3, Remark 6.10]).

Kapitel 3

A posteriori Fehlerschätzung

In diesem Kapitel wollen wir a posteriori Fehlerschätzer in der Energienorm $|||\cdot|||$ entwickeln. Dabei wird einerseits der $h - h/2$-Fehlerschätzer η_H in Anlehnung an Ferraz-Leite/Praetorius 2007 [16] dargestellt. Andererseits werden wir auch einen sogenannten Glättungsschätzer η_M betrachten. Die Analysis für η_M wird in sehr allgemeiner Form in Carstensen/Praetorius 2006 [8] geführt. Auf diese Arbeit gehen auch die Lokalisierungstechniken, die in Abschnitt 3.1 studiert werden, zurück. Wir halten uns in der Analysis von η_M an die Arbeit Ferraz-Leite/Funken/Praetorius 2007 [15].

Ein Fehlerschätzer η heißt effizient, falls es eine Konstante C_{eff} gibt, sodass

$$C_{\text{eff}}^{-1}\eta \leq \|\phi - \phi_h\|$$

gilt. Er heißt zuverlässig, falls es eine Konstante C_{rel} gibt, sodass

$$|||\phi - \phi_h||| \leq C_{\text{rel}}\eta$$

gilt. Die Konstanten sollten jeweils von der Anzahl der Elemente, den lokalen Netzweiten sowie der diskreten Lösung ϕ_h unabhängig sein. Eine Norm wird von uns als lokal bezeichnet, falls für eine Triangulierung $\mathcal{T}$ und einen Raum X die Beziehung

$$\|\cdot\|^2_{X(\Gamma)} \sim \sum_{T_j \in \mathcal{T}} \|\cdot\|^2_{X(T_j)}$$

gilt. Beispiele für lokale Normen in diesem Sinne sind die L^2-Norm und die H^1-Norm. Die Energienorm ist nicht lokal. Die Resultate des folgenden Abschnitts 3.1, eine inverse Abschätzung aus Graham/Hackbusch/Sauter 2005 [19] und eine Approximationsaussage aus Carstensen/Praetorius 2006 [8] dienen zur Lokalisierung von in der Energienorm gegebenen Schätzern.

Sowohl die Berechnung von η_H, als auch die von η_M erfordert es, zwei Triangulierungen zu betrachten. Obwohl die Glättungstechniken nur wenig Zusammenhang zwischen den beiden beteiligten Netzen fordern, erweist es sich als für die Analysis und die Implementierung sinnvoll, stets mit zwei Paneelierungen $\mathcal{T}_h$ und $\mathcal{T}_{h/2}$ zu arbeiten, wobei $\mathcal{T}_{h/2}$ aus $\mathcal{T}_h$ durch einmalige uniforme Verfeinerung hervorgeht.

Wir werden zeigen, dass beide Arten von Fehlerschätzern, die in dieser Arbeit behandelt werden, mit gewissen Einschränkungen zuverlässig und effizient sind. Darüberhinaus zeigen jüngste Ergebnisse in Erath/Ferraz-Leite/Funken/Praetorius 2007 [12], dass η_H und η_M stets äquivalent sind.

3.1 Lokalisierungstechniken

Ziel ist es, aus einem Fehlerschätzer der Gestalt

$$|||\phi_h - \phi_H|||$$

einen lokalen Fehlerschätzer in der L^2-Norm zu gewinnen. Dabei sind ϕ_h und ϕ_H zu verschiedenen Netzen $\mathcal{T}_h$ und $\mathcal{T}_H$ berechnete diskrete Lösungen mit potentiell verschiedenem Polynomgrad. Wir werden sehen, dass man aus einem solchen Schätzer mit Hilfe einer lokalisierten inversen Abschätzung und einer Approximationsaussage einen äquivalenten Schätzer in einer gewichteten L^2-Norm konstruieren kann.

Die Approximationseigenschaft der L^2-Projektion wird in Carstensen/Praetorius 2006 [8, Theorem 4.1] bewiesen. Allerdings wird der Satz für beliebige diskrete Räume X mit $\mathcal{P}^0(\mathcal{T}) \subseteq X$ formuliert. Die Argumente des Beweises gelten jedoch nur für echt lokale Räume. Daher beschränken wir uns auf die Formulierung für Räume $X = \mathcal{P}^k(\mathcal{T})$.

Satz 3.1 (Approximationseigenschaft der L^2-Projektion). *Sei $X_h = \mathcal{P}^k(\mathcal{T})$ mit $k \geq 0$. Dann erfüllt die L^2-Projektion $\Pi : L^2(\Gamma) \longrightarrow X_h$ die Approximationseigenschaft*

$$\|(\mathbb{I} - \Pi)v\|_{\widetilde{H}^{-\alpha}(\Gamma)} \leq \widetilde{C}_{apx} \min\{\|h^\alpha v\|_{L^2(\Gamma)}, \|h^\alpha(\mathbb{I} - \Pi)v\|_{L^2(\Gamma)}\} \quad \forall v \in L^2(\Gamma). \tag{3.1}$$

Die Konstante $\widetilde{C}_{apx}$ ist unabhängig von der Anzahl und der Größe der Elemente in $\mathcal{T}_h$ und hängt nur von Γ ab.

Beweis. ***1. Schritt.*** Sei $\mathcal{T} = \{T_1, \ldots, T_n\}$ eine Triangulierung des Randes Γ und bezeichne $\Pi_0 : L^2(\Gamma) \longrightarrow \mathcal{P}^0(\mathcal{T}_h)$ die L^2-Projektion auf $\mathcal{P}^0(\mathcal{T}_h)$. Für beliebiges $T_j \in \mathcal{T}$ und $u \in H^1(T_j)$ gilt

$$\|(\mathbb{I} - \Pi)u\|_{L^2(T_j)} = \inf_{v_h \in X_h} \|u - v_h\|_{L^2(T_j)} \leq \inf_{v_h \in \mathcal{P}^0(\mathcal{T}_h)} \|u - v_h\|_{L^2(T_j)} = \|(\mathbb{I} - \Pi_0)u\|_{L^2(T_j)}.$$

Gemäß der Poincaré-Ungleichung (2.1) gilt

$$\|(\mathbb{I} - \Pi_0)u\|_{L^2(T_j)} \leq C h_j \|\nabla u\|_{L^2(T_j)}.$$

Daher hat der Operator $T_\alpha : H^\alpha(T_j) \longrightarrow L^2(T_j), u \longmapsto u - \Pi u$ die Operatornorm $\|T_0\| = 1$ und $\|T_1\| \leq C h_j$. Laut Interpolationsabschätzung für Operatornormen (2.2) folgt $\|T_\alpha\| \leq \|T_0\|^{1-\alpha}\|T_1\|^\alpha \leq C^\alpha h_j^\alpha$ und somit insgesamt

$$\|(\mathbb{I} - \Pi)u\|_{L^2(T_j)} \leq C^\alpha h_j^\alpha \|u\|_{H^\alpha(T_j)} \tag{3.2}$$

für $u \in H^\alpha(T_j)$. Auf Grund der Symmetrie von Orthogonalprojektionen ist

$$\langle(\mathbb{I} - \Pi)v, u\rangle = \langle v, (\mathbb{I} - \Pi)u\rangle \leq \sum_{j=1}^{n} \|v\|_{L^2(T_j)} \|(\mathbb{I} - \Pi)u\|_{L^2(T_j)} \tag{3.3}$$

für beliebiges $v \in L^2(\Gamma)$ und $u \in H^\alpha(\Gamma)$. Die Kombination der Gleichungen (3.2) und (3.3) sowie das Anwenden des Lemmas 2.11 beweist schließlich

$$\langle(\mathbb{I} - \Pi)v, u\rangle \leq C^\alpha \sum_{j=1}^{n} \|h^\alpha v\|_{L^2(T_j)} \|u\|_{H^\alpha(T_j)} \leq C^\alpha \|h^\alpha v\|_{L^2(\Gamma)} \|u\|_{H^\alpha(\Gamma)}.$$

2. Schritt. Wegen $\Pi^2 = \Pi$ folgt für $w := (\mathbb{I} - \Pi)v$, dass $(\mathbb{I} - \Pi)w = w = (\mathbb{I} - \Pi)v$ ist. Wendet man den ersten Schritt des Beweises auf w an, so erhält man unmittelbar

$$\|(\mathbb{I} - \Pi)v\|_{\widetilde{H}^{-\alpha}} = \|(\mathbb{I} - \Pi)w\|_{\widetilde{H}^{-\alpha}(\Gamma)} \leq C^\alpha \|h^\alpha w\|_{L^2(\Gamma)} = C^\alpha \|h^\alpha(\mathbb{I} - \Pi)v\|_{L^2(\Gamma)}.$$

Mit $\widetilde{C}_{\text{apx}} = C^{\alpha}$ ist der Satz somit vollständig bewiesen. ■

Bemerkung. In Satz 3.1 gilt

$$\|h^{\alpha}(\mathbb{I} - \Pi)v\|_{L^2(\Gamma)} \leq \|h^{\alpha}v\|_{L^2(\Gamma)},$$

sodass das Minimum stets mit $\|h^{\alpha}(\mathbb{I} - \Pi)v\|_{L^2(\Gamma)}$ angenommen wird.

Aus der Äquivalenz der Normen

$$|\!|\!| \cdot |\!|\!| \leq C\| \cdot \|_{\widetilde{H}^{-1/2}(\Gamma)}$$

folgt unmittelbar das folgende Resultat.

Korollar 3.2. *Für $\alpha = 1/2$ und unter den Voraussetzungen von Satz 3.1 besitzt die L^2-Projektion $\Pi : \widetilde{H}^{-1/2}(\Gamma) \to X_h$ die Approximationseigenschaft*

$$|\!|\!|(\mathbb{I} - \Pi)v|\!|\!| \leq C_{apx} \min\{\|h^{\alpha}v\|_{L^2(\Gamma)}, \|h^{\alpha}(\mathbb{I} - \Pi)v\|_{L^2(\Gamma)}\} \quad \forall v \in L^2(\Gamma) \tag{3.4}$$

in der Energienorm mit $C_{apx} = C\widetilde{C}_{apx}$. ■

Korollar 3.3 (Approximationseigenschaft der Galerkinprojektion). *Für $\alpha = 1/2$ und unter den Voraussetzungen von Satz 3.1 besitzt die Galerkinprojektion $\mathbb{G} : \widetilde{H}^{-1/2}(\Gamma) \longrightarrow X_h$ die Approximationseigenschaft*

$$|\!|\!|(\mathbb{I} - \mathbb{G})v|\!|\!| \leq C_{apx} \min\{\|h^{\alpha}v\|_{L^2(\Gamma)}, \|h^{\alpha}(\mathbb{I} - \mathbb{G}))v\|_{L^2(\Gamma)}\} \quad \forall v \in L^2(\Gamma). \tag{3.5}$$

Beweis. ***1. Schritt.*** Die Bestapproximationseigenschaft der Galerkinprojektion impliziert

$$|\!|\!|(\mathbb{I} - \mathbb{G})v|\!|\!| \leq |\!|\!|(\mathbb{I} - \Pi)v|\!|\!|.$$

Die Äquivalenz der Normen $|\!|\!| \cdot |\!|\!| \leq C\| \cdot \|_{\widetilde{H}^{-1/2}(\Gamma)}$ sowie die Approximationseigenschaft der L^2-Projektion beweisen

$$|\!|\!|(\mathbb{I} - \mathbb{G})v|\!|\!| \leq |\!|\!|(\mathbb{I} - \Pi)v|\!|\!| \leq C_{\text{apx}}\|h^{1/2}v\|_{L^2(\Gamma)}.$$

2. Schritt. Wir setzen $w = (\mathbb{I} - \mathbb{G})v$ und halten fest, dass $\mathbb{I} - \mathbb{G}$ eine Projektion ist und somit $(\mathbb{I} - \mathbb{G})w = w$ gilt. Wir wenden Schritt 1 auf w an, um

$$|\!|\!|(\mathbb{I} - \mathbb{G})v|\!|\!| = |\!|\!|(\mathbb{I} - \mathbb{G})w|\!|\!| \leq C_{\text{apx}}\|h^{1/2}w\|_{L^2(\Gamma)} = C_{\text{apx}}\|(\mathbb{I} - \mathbb{G})v\|_{L^2(\Gamma)}$$

zu erhalten. Insgesamt haben wir die Behauptung

$$|\!|\!|(\mathbb{I} - \mathbb{G})v|\!|\!| \leq C_{\text{apx}} \min\{\|h^{1/2}v\|_{L^2(\gamma)}, \|h^{1/2}(\mathbb{I} - \mathbb{G})v\|_{L^2(\Gamma)}\}$$

gezeigt. ■

Nun wird eine inverse Abschätzung zitiert, die nur lokale Kenngrößen des Netzes in die Ungleichung einfließen lässt. Üblicherweise verwenden inverse Abschätzungen die globale Kenngröße $h_{\min}$ und sind daher für die Lokalisierung von Fehlerschätzern ungeeignet. Um die lokalisierte inverse Abschätzung zeigen zu können, benötigt man schwache Voraussetzungen an das Netz $\mathcal{T}$, die sogar anisotrope Netze mit Elementen unterschiedlicher Form zulassen. Wir passen die Formulierung des Satzes an unsere Notation an.

Satz 3.4 (Graham/Hackbusch/Sauter 2005 [19, Theorem 3.6]). *Für ein fastreguläres Netz $\mathcal{T}$ gibt es eine Konstante $\widetilde{C}_{inv}$, sodass für $0 \le s \le 1$ und $-a \le \alpha \le a \in \mathbb{R}$*

$$\|\varrho^{s+\alpha} v_h\|_{L^2(\Gamma)} \le \widetilde{C}_{inv} \|\varrho^{\alpha} v_h\|_{\widetilde{H}^{-s}(\Gamma)} \quad \forall v_h \in \mathcal{P}^p(\mathcal{T}) \tag{3.6}$$

gilt. Die Konstante $\widetilde{C}_{inv}$ hängt nur von Γ, p, s, der Form der Elemente in $\mathcal{T}$ sowie den Konstanten $\kappa(\mathcal{T}), \widehat{\kappa}(\mathcal{T})$ ab, jedoch nicht von der Anzahl oder der Größe der Elemente. ■

Korollar 3.5 (Lokale inverse Abschätzung). *Für ein fastreguläres Netz $\mathcal{T}$ gibt es eine Konstante C_{inv}, sodass*

$$\|\varrho^{1/2} v_h\|_{L^2(\Gamma)} \le C_{inv} |\!|\!| v_h |\!|\!| \quad \forall v_h \in \mathcal{P}^p(\mathcal{T}) \tag{3.7}$$

gilt. Die Konstante ergibt sich aus der Äquivalenz der Normen $\|\cdot\|_{\widetilde{H}^{-1/2}(\Gamma)} \le C |\!|\!| \cdot |\!|\!|$ sowie der Approximationskonstanten aus Satz 3.4 zu $C_{inv} = C \cdot \widetilde{C}_{inv}$. ■

3.2 Der $h - h/2$-Schätzer

In diesem Abschnitt führen wir den Fehlerschätzer η_H ein und zeigen wie in Ferraz-Leite/Praetorius 2007 [16], dass er stets effizient mit konstante $C_{\text{eff}} = 1$ ist. Die Zuverlässigkeit von η_H wird unter der Saturationsannahme bewiesen.

Satz 3.6. *Der $h - h/2$-Fehlerschätzer*

$$\eta_H := |\!|\!| \phi_{h/2} - \phi_h |\!|\!|$$

ist stets effizient mit $C_{eff} = 1$. Unter der Saturationsannahme

$$|\!|\!| \phi - \phi_{h/2} |\!|\!| \le q |\!|\!| \phi - \phi_h |\!|\!| \tag{3.8}$$

mit $q \in (0,1)$ gleichmäßig beschränkt für $h \to 0$ ist η_H auch zuverlässig mit $C_{rel} = (1-q^2)^{-1/2}$.

Beweis. ***1. Schritt.*** *Effizienz von η_H:*
Wegen $X_h \subseteq X_{h/2}$ gilt $\phi_{h/2} - \phi_h \in X_{h/2}$. Daher folgt aus der Galerkinorthogonalität, dass $\phi - \phi_{h/2}$ orthogonal zu $\phi_{h/2} - \phi_h$ ist. Wir wenden den Satz von Pythagoras an, um

$$|\!|\!| \phi - \phi_h |\!|\!|^2 = |\!|\!| \phi - \phi_{h/2} |\!|\!|^2 + |\!|\!| \phi_{h/2} - \phi_h |\!|\!|^2$$

bzw. umgeschrieben

$$\eta_H^2 = |\!|\!| \phi - \phi_h |\!|\!|^2 - |\!|\!| \phi - \phi_{h/2} |\!|\!|^2 \le |\!|\!| \phi - \phi_h |\!|\!|^2$$

zu erhalten.
2. Schritt. *Zuverlässigkeit von η_H:*
Mit derselben Argumentation wie in Schritt 1 erhält man

$$|\!|\!| \phi - \phi_h |\!|\!|^2 = |\!|\!| \phi - \phi_{h/2} |\!|\!|^2 + |\!|\!| \phi_{h/2} - \phi_h |\!|\!|^2 = |\!|\!| \phi - \phi_{h/2} |\!|\!|^2 + \eta_H^2 \le q^2 |\!|\!| \phi - \phi_h |\!|\!|^2 + \eta_H^2.$$

Insbesondere gilt

$$\sqrt{1-q^2} |\!|\!| \phi - \phi_h |\!|\!| \le \eta_H$$

und somit

$$|||\phi - \phi_h||| \leq \frac{1}{\sqrt{1-q^2}}\eta_H,$$

womit die Zuverlässigkeit gezeigt ist. ■

Bemerkung. In den numerischen Experimenten in Kapitel 5 wird die Gültigkeit der Saturationsannahme (3.8) beobachtet. Die Annahme wird in Dörfler/Nochetto 2002 [11] für die Finite Elemente Methode unter schwachen Bedingungen an die Netzverfeinerung bewiesen.

Bemerkung. Aufgrund der Bestapproximationseigenschaft der Galerkinprojektion gilt

$$|||\phi - \phi_{h/2}||| \leq |||\phi - \phi_h|||.$$

Die Saturationsannahme (3.8) ist eine echte Verschärfung dieser Aussage in der eine gleichmäßige Verbesserung der Approximation gefordert wird.

3.2.1 Lokalisierung des $h - h/2$-Schätzers

Der Fehlerschätzer η_H kann dazu dienen den Fehler in der Energienorm zu schätzen. Ein Algorithmus, der als Abbruchkriterium $\eta_H < \varepsilon$ besitzt, wird jedoch üblicherweise $\phi_{h/2}$ als Lösung zurückgeben, da für ϕ_h und $\phi_{h/2}$ aufgrund der Bestapproximationseigenschaft und $X_h \subseteq X_{h/2}$ die Ungleichung $|||\phi - \phi_{h/2}||| \leq |||\phi - \phi_h|||$ gilt. Unter diesem Gesichtspunkt kann die Berechnung von ϕ_h als Zwischenschritt zur Berechnung von η_H interpretiert werden. Man beachte, dass die Systemmatrix zum Ermitteln von ϕ_h nicht aufgestellt werden muss. Da die Räume $X_h \subseteq X_{h/2}$ eingebettet sind, kann man über geeignete Nebenbedingungen die Systemmatrix zur Ermittlung von $\phi_{h/2}$ auch für die Berechnung von ϕ_h verwenden. Dennoch ist das Lösen eines vollbesetzten Gleichungssystems vergleichsweise rechenintensiv. Wir werden sehen, dass man einen äquivalenten Schätzer $\widetilde{\eta}_H$ konstruieren kann, der ohne die exakte Berechnung von ϕ_h auskommt.

Außerdem ist η_H ungeeignet, um einen adaptiven Netzverfeinerungsalgorithmus zu steuern, da die Energienorm nicht lokal ist und daher nicht auf einzelnen Paneelen über die lokale Abweichung der diskreten Lösung entschieden werden kann. Um einen zu η_H äquivalenten lokalen Schätzer μ_H in der L^2-Norm zu konstruieren, werden wir die Techniken aus Abschnitt 3.1 anwenden.

Die Ideen, die wir verwenden, stammen ursprünglich aus Carstensen/Praetorius 2006 [8]. Es wird der knapp formulierte Beweis von Ferraz-Leite/Praetorius 2007 [16] ausführlich dargelegt. Dabei werden [16, Theorem 3.2] und [16, Theorem 3.4] übersichtlich und kompakt zum folgenden Satz zusammengefasst.

Satz 3.7. *Mit $\Pi_h : L^2(\Gamma) \to \mathcal{P}^0(\mathcal{T}_h)$ der L^2-Projektion auf $\mathcal{P}^0(\mathcal{T}_h)$ definieren wir die Fehlerschätzer*

- $\mu_H := \|\varrho^{1/2}(\phi_{h/2} - \phi_h)\|_{L^2(\Gamma)}$,
- $\widetilde{\eta}_H := |||\phi_{h/2} - \Pi_h\phi_{h/2}|||$,
- $\widetilde{\mu}_H := \|\varrho^{1/2}(\phi_{h/2} - \Pi_h\phi_{h/2})\|_{L^2(\Gamma)}$.

Mit der Approximationskonstante C_{apx} aus Korollar 3.2 und der Konstanten C_{inv} der inversen Abschätzung aus Korollar 3.5 gelten

$$\eta_H \leq \widetilde{\eta}_H \leq C_{apx}\sigma(\mathcal{T}_h)^{1/2}\widetilde{\mu}_H \quad \textit{und} \quad \widetilde{\mu}_H \leq \mu_H \leq \sqrt{2}C_{inv}\eta_H. \tag{3.9}$$

Insbesondere sind alle eingeführten Fehlerschätzer stets effizient. Unter der Saturationsannahme (3.8) *sind sie auch zuverlässig. Die Zuverlässigkeit von μ_H und $\widetilde{\mu}_H$ sowie die Effizienz von $\widetilde{\eta}_H$ hängen von der Formregularität des Netzes $\sigma(\mathcal{T}_h)$ ab.*

Beweis. ***1. Schritt.*** *$\eta_H \leq \widetilde{\eta}_H$:*
Aus der Bestapproximationseigenschaft der Galerkinprojektion sowie der Einbettung der Räume $\mathcal{P}^0(\mathcal{T}_h) \subseteq \mathcal{P}^0(\mathcal{T}_{h/2})$ schließt man

$$\eta_H = |||\phi_{h/2} - \phi_h||| = |||(\mathbb{I} - \mathbb{G}_h)\phi_{h/2}||| \leq |||(\mathbb{I} - \Pi_h)\phi_{h/2}||| = \widetilde{\eta}_H.$$

2. Schritt. *$\widetilde{\eta}_H \leq C_{apx}\sigma(\mathcal{T}_h)\widetilde{\mu}_H$:*
Aus der Approximationseigenschaft der L^2-Projektion folgt unmittelbar

$$\widetilde{\eta}_H = |||(\mathbb{I} - \Pi_h)\phi_{h/2}||| \leq C_{\mathrm{apx}}\|h^{1/2}(\mathbb{I} - \Pi_h)\phi_{h/2}\| \leq C_{\mathrm{apx}} \underbrace{\|(h/\varrho)^{1/2}\|_{L^\infty(\Gamma)}}_{=\sigma(\mathcal{T}_h)^{1/2}} \widetilde{\mu}_H.$$

3. Schritt. *$\widetilde{\mu}_H \leq \mu_H$:*
Aufgrund der speziellen Wahl des Ansatzraumes $X_h = \mathcal{P}^0(\mathcal{T}_h)$ liefert die L^2-Projektion Π_h die Bestapproximation bezüglich der L^2-Norm auf jedem einzelnen Randelement. Somit ist

$$\widetilde{\mu}_H^2 = \sum_{T_j \in \mathcal{T}_h} \varrho_{T_j}\|(\mathbb{I} - \Pi_h)\phi_{h/2}\|^2_{L^2(T_j)} \leq \sum_{T_j \in \mathcal{T}_h} \varrho_{T_j}\|(\mathbb{I} - \mathbb{G}_h)\phi_{h/2}\|^2_{L^2(T_j)} = \mu_H^2.$$

4. Schritt. *$\mu_H \leq \sqrt{2}C_{inv}\eta_H$:*
Wir wenden die lokalisierte inverse Abschätzung (3.7) an und erhalten

$$\|\varrho^{1/2}(\phi_h - \phi_{h/2})\|_{L^2(\Gamma)} = \sqrt{2}\|(\varrho/2)^{1/2}(\phi_h - \phi_{h/2})\|_{L^2(\Gamma)} \leq \sqrt{2}C_{\mathrm{inv}}|||\phi_h - \phi_{h/2}|||$$

aufgrund von $\phi_h - \phi_{h/2} \in \mathcal{P}^0(\mathcal{T}_{h/2})$. ■

Die Tatsache, dass die L^2-Norm lokal ist, d.h. $\|\cdot\|^2_{L^2(\Gamma)} = \sum_{j=1}^n \|\cdot\|^2_{T_j}$, ermöglicht es den Schätzer $\widetilde{\mu}_H$, genauer die Beiträge $\widetilde{\mu}_{H,j} := \|\varrho^{1/2}(\phi_{h/2} - \Pi_h\phi_{h/2})\|_{L^2(T_j)}$, als Verfeinerungsindikatoren eines adaptiven isotropen Netzverfeinerungsalgorithmus zu verwenden. Ebenso wäre der Einsatz von μ_H denkbar, der jedoch einen höheren Rechenaufwand erfordert.

Man beachte, dass die L^2-Projektion Π_h auf $\mathcal{P}_0(\mathcal{T}_h)$ mit $\mathcal{T}_h = \{T_1, \ldots, T_n\}$ durch

$$(\Pi_h v)\,|_\tau = \frac{1}{|\tau|}\int_\tau v\,ds \quad \text{für alle } v \in L^2(\Gamma) \text{ und } \tau \in \mathcal{T}_h$$

gegeben ist. Diese Tatsache folgt aus der Orthogonalitätseigenschaft

$$0 = (v - \Pi_h v; \chi_T)_{L^2(\Gamma)} = \int_T v\ ds - \int_T (\Pi_h v) ds = \int_T v\ ds - |T|(\Pi_h v)|_T \quad \forall T \in \mathcal{T}_h.$$

Insbesondere lässt sich der Schätzer $\widetilde{\mu}_H$ in linearem Aufwand $\mathcal{O}(n)$ mit n der Anzahl der Randelemente berechnen und fällt somit nicht ins Gewicht.

3.3 Glättungstechniken

3.3.1 Abstrakte Analysis

In diesem Abschnitt soll das Konzept der Glättungstechniken in einem möglichst allgemeinen Rahmen wie in Erath/Ferraz-Leite/Funken/Praetorius 2007 [12] erklärt werden. Die ursprünglichen

Ideen gehen auf CARSTENSEN/PRAETORIUS 2006 [8] zurück, jedoch sind die Formulierungen in der neueren Arbeit didaktisch besser strukturiert. Die Voraussetzungen für die Zuverlässigkeit und die Effizienz werden scharf abgegrenzt und im Vergleich zur ursprünglichen Darstellung abgeschwächt. Die nur sehr knapp geführten Beweise in ERATH/FERRAZ-LEITE/FUNKEN/PRAETORIUS 2007 [12] werden vervollständigt.

Es bezeichne X_H einen diskreten Raum mit Polynomen niedriger Ordnung auf einem feinen Netz $\mathcal{T}_H$. Der diskrete Raum X_h möge Polynome höheren Grades enthalten, also zur Approximation glatter Funktionen besser geeignet sein, und sei über einem groben Netz $\mathcal{T}_h$ definiert. Die entsprechenden Galerkinprojektionen werden mit $\mathbb{G}_h : \widetilde{H}^{-1/2}(\Gamma) \longrightarrow X_h$ bzw. $\mathbb{G}_H : \widetilde{H}^{-1/2}(\Gamma) \longrightarrow X_H$ bezeichnet, die zugehörigen Galerkinlösungen mit ϕ_h bzw. ϕ_H.

Satz 3.8 (Hauptsatz der Glättungstechniken). *Seien*

$$q := \max_{v_h \in X_h \setminus \{0\}} \min_{v_H \in X_H \setminus \{0\}} \frac{|||v_h - v_H|||}{|||v_h|||}$$

und

$$\delta_{hH} := \frac{|||\phi - \mathbb{G}_h\phi|||}{|||\phi - \mathbb{G}_H\phi|||}.$$

Der Fehlerschätzer

$$\eta_M := |||\phi_H - \mathbb{G}_h\phi_H|||$$

erfüllt

$$\eta_M \leq (1 + \delta_{hH})|||\phi - \phi_H|||$$

und ist somit effizient. Ist außerdem $q^2 + \delta_{hH}^2 < 1$, so ist η_M auch zuverlässig

$$|||\phi - \phi_H||| \leq \frac{1}{\sqrt{1 - q^2} - \delta_{hH}} \eta_M.$$

Beweis. ***1. Schritt.*** *Effizienz von η_M:*
Durch Anwenden der Dreiecksungleichung sowie Berücksichtigung, dass $\mathbb{I} - \mathbb{G}_H$ eine Orthogonalprojektion ist, erhält man

$$\eta_M = |||(\mathbb{I} - \mathbb{G}_h)\phi_H||| \leq |||(\mathbb{I} - \mathbb{G}_h)(\phi - \phi_H)||| + |||(\mathbb{I} - \mathbb{G}_h)\phi||| \leq |||\phi - \phi_H||| + \delta_{hH}|||\phi - \mathbb{G}_H\phi|||.$$

2. Schritt. *Zuverlässigkeit von η_M:*
Zunächst halten wir fest, dass mit $q^2 + \delta_{hH}^2 < 1$ wegen

$$\delta_{hH}^2 < 1 - q^2 \quad \Rightarrow \quad \delta_{hH} < \sqrt{1 - q^2}$$

sowohl $\frac{\delta_{hH}}{\sqrt{1-q^2}} < 1$ als auch $\sqrt{1 - q^2} - \delta_{hH} > 0$ folgen. Wir definieren $e := \phi - \phi_H$ und $e_h := \mathbb{G}_h e$. Im Folgenden ist $v_H \in X_H$ beliebig. Wir verwenden, dass $\mathbb{G}_h$ als Orthogonalprojektion selbstadjungiert ist, sowie die Galerkinorthogonalität in den Räumen X_h und X_H, um

$$\begin{aligned} |||e_h|||^2 = \langle\!\langle e_h, e_h \rangle\!\rangle = \langle\!\langle \mathbb{G}_h e, e_h \rangle\!\rangle &= \langle\!\langle e, \mathbb{G}_h e_h \rangle\!\rangle \\ &\leq \langle\!\langle e, e_h - v_H \rangle\!\rangle \\ &\leq |||e|||\,|||e_h - v_H||| \end{aligned}$$

abzuschätzen. Bilden des Minimums über alle $v_H \in X_H$ ergibt

$$|||e_h|||^2 \leq |||e|||q|||e_h|||.$$

Insgesamt erhalten wir

$$|||e_h||| \leq q|||e|||.$$

Der Satz von Pythagoras impliziert

$$|||e|||^2 = |||e - e_h|||^2 + |||e_h|||^2$$

und dadurch

$$(1 - q^2)|||e|||^2 \leq |||e - e_h|||^2.$$

Mit der Dreiecksungleichung erhalten wir

$$\begin{aligned} |||e||| \leq \frac{1}{\sqrt{1-q^2}}|||e - e_h||| &= \frac{1}{\sqrt{1-q^2}}|||(\mathbb{I} - \mathbb{G}_h)(\phi - \phi_H)||| \\ &\leq \frac{1}{\sqrt{1-q^2}}(|||(\mathbb{I} - \mathbb{G}_h)\phi||| + |||(\mathbb{I} - \mathbb{G}_h)\phi_H|||). \end{aligned}$$

Mit der Definition von δ_{hH} gilt also

$$\begin{aligned} |||\phi - \phi_H||| &\leq \frac{1}{\sqrt{1-q^2}}\eta_M + \frac{\delta_{hH}}{\sqrt{1-q^2}}|||\phi - \phi_H||| \\ \Big(1 - \frac{\delta_{hH}}{\sqrt{1-q^2}}\Big)|||\phi - \phi_H||| &\leq \frac{1}{\sqrt{1-q^2}}\eta_M. \end{aligned}$$

Der Faktor auf der linken Seite der letzten Ungleichung lässt sich zu $\frac{\sqrt{1-q^2}-\delta_{hH}}{\sqrt{1-q^2}}$ umschreiben, womit nach Multiplikation mit dem Kehrwert die Behauptung des Satzes folgt. ■

Bemerkung. In Carstensen/Praetorius 2006 [8] wird die asymptotische Zuverlässigkeit von η_M unter der stärkeren Voraussetzung $\delta_{hH} \to 0$ für $h \to 0$ gezeigt. Diese Forderung wird als Annahme über Terme höherer Ordnung bezeichnet, da sie besagt, dass $|||\phi - \mathbb{G}_h\phi|||$ von höherer Ordnung als $|||\phi - \mathbb{G}_H\phi|||$ ist.

3.3.2 η_M für die Symmsche Integralgleichung

Wir zeigen jetzt wie in Ferraz-Leite/Funken/Praetorius 2007 [15], dass η_M unter gewissen Voraussetzungen im Falle der Symmschen Integralgleichung zuverlässig und effizient ist. Wir gehen davon aus, dass das feine Netz $\mathcal{T}_H$ durch ℓ-malige uniforme Verfeinerung aus dem Netz $\mathcal{T}_h$ hervorgeht. In diesem Fall ist $\|H/h\|_{L^\infty(\Gamma)} = 2^{-\ell}$. Wir legen die diskreten Räume durch $X_h = \mathcal{P}^1(\mathcal{T}_h)$ und $X_H = \mathcal{P}^0(\mathcal{T}_H)$ fest.

Satz 3.9. *Seien $\phi \in H^{1+\varepsilon}(\Gamma)$ für ein $\varepsilon > 0$ und $\ell > -2\log_2(C_{apx}^{-1}C_{inv}^{-1}\sigma(\mathcal{T}_h)^{-1/2})$ mit den Konstanten aus Korollar 3.2 und Korollar 3.5, dann ist der Schätzer η_M asymptotisch zuverlässig.*

Beweis. ***1. Schritt.*** *Wir zeigen $q < 1$:*
Die lokale inverse Abschätzung (3.7) zeigt für $v_h \in X_h$

$$\|h^{1/2}v_h\|_{L^2(\Gamma)} = \|(h/\varrho)^{1/2}\|_{L^\infty(\Gamma)}\|\varrho^{1/2}v_h\|_{L^2(\Gamma)} \leq C_{\mathrm{inv}}\sigma(\mathcal{T}_h)^{1/2}|\!|\!|v_h|\!|\!|. \tag{3.10}$$

Die Approximationseigenschaft der L^2-Projektion (3.4) impliziert für $v \in \widetilde{H}^{-1/2}(\Gamma)$ und mit Π_H der L^2-Projektion auf X_H

$$|\!|\!|v - \Pi_H v|\!|\!| \leq C_{\mathrm{apx}}\|H^{1/2}v\|_{L^2(\Gamma)}. \tag{3.11}$$

Das Kombinieren der Gleichungen (3.10) und (3.11) ergibt insgesamt

$$\begin{aligned}\min_{v_H \in X_H} |\!|\!|v_h - v_H|\!|\!| \leq |\!|\!|v_h - \Pi_H v_h|\!|\!| \leq C_{\mathrm{apx}}\|H^{1/2}v_h\|_{L^2(\Gamma)} &= C_{\mathrm{apx}}\|(H/h)^{1/2}\|_{L^\infty(\Gamma)}\|h^{1/2}v_h\|_{L^2(\Gamma)} \\ &\leq C_{\mathrm{apx}}C_{\mathrm{inv}}2^{-\ell/2}\sigma(\mathcal{T}_h)^{1/2}|\!|\!|v_h|\!|\!|.\end{aligned}$$

Nach Dividieren der Ungleichung durch $|\!|\!|v_h|\!|\!|$ und Bilden des Maximums folgt

$$q = \max_{v_h \in X_h\setminus\{0\}} \min_{v_H \in X_H} \frac{|\!|\!|v_h - v_H|\!|\!|}{|\!|\!|v_h|\!|\!|} \leq C_{\mathrm{apx}}C_{\mathrm{inv}}2^{-\ell/2}\sigma(\mathcal{T}_h)^{1/2}.$$

Für $\ell > -2\log_2(C_{\mathrm{apx}}^{-1}C_{\mathrm{inv}}^{-1}\sigma(\mathcal{T}_h)^{-1/2})$ ist $q < 1$.
2. Schritt. *Wir zeigen $\delta_{hH} \to 0$ für $h \to 0$:*
Laut den Bestapproximationsaussagen

$$|\!|\!|\phi - \mathbb{G}_h\phi|\!|\!| = \mathcal{O}(h^{3/2}) \quad \text{sowie} \quad |\!|\!|\phi - \mathbb{G}_H\phi|\!|\!| = \mathcal{O}(H^{3/2+\varepsilon})$$

aus Korollar 2.22, ist $|\!|\!|\phi - \mathbb{G}_h\phi|\!|\!|$ von höherer Ordnung als $|\!|\!|\phi - \mathbb{G}_H\phi|\!|\!|$, was äquivalent zu $\delta_{hH} \to 0$ ist. Da wir insgesamt $q < 1$ und $\delta_{hH} \to 0$ gezeigt haben, folgt asymptotisch $q^2 + \delta_{hH}^2 < 1$. ∎

Bemerkung. In Praxis ist nicht bestimmbar, wie groß ℓ gewählt werden soll. Allerdings beobachtet man im numerischen Experiment, dass einmalige uniforme Verfeinerung, hinreichend ist. Durch den Einfluss der Formregularität ergibt sich eine Einschränkung auf isotrope Netze. Allerdings zeigen numerische Experimente, dass auch für anisotrope Netzfolgen, die aus fastregulären Netzen bestehen, der vorgestellte Schätzer effizient und zuverlässig ist.

3.3.3 Lokalisierung von η_M

So wie auch schon der Fehlerschätzer η_H ist auch η_M nicht lokal und kann somit nicht als Indikator für einen Netzverfeinerungsalgorithmus verwendet werden. Außerdem bedingt die Berechnung von $(\mathbb{I} - \mathbb{G}_H)\phi_h$ das Lösen eines Gleichungssystems mit einer vollbesetzten Matrix. Allerdings kann man

den Schätzer analog wie in Abschnitt 3.2.1 lokalisieren und erhält einen äquivalenten Schätzer, der statt der Galerkinprojektion lediglich die weitaus weniger rechenintensive L^2-Projektion verwendet. Wir fassen sie Ergebnisse aus Ferraz-Leite/Funken/Praetorius 2007 [15] kompakt in einem Satz zusammen und vervollständigen den Beweis.

Satz 3.10. *Sei $\mathcal{T}_H$ das durch ℓ uniforme Verfeinerungen aus $\mathcal{T}_h$ hervorgegangene Netz. Mit $\Pi_h : L^2(\Gamma) \longrightarrow \mathcal{P}^1(\mathcal{T}_h)$ der L^2-Projektion auf $\mathcal{P}^1(\mathcal{T}_h)$ definieren wir die Fehlerschätzer*

- $\mu_M = \|\varrho^{1/2}(\mathbb{I} - \mathbb{G}_h)\phi_H\|_{L^2(\Gamma)}$,
- $\widetilde{\eta}_M = |\!|\!|(\mathbb{I} - \Pi_h)\phi_H|\!|\!|$,
- $\widetilde{\mu}_M = \|\varrho^{1/2}(\mathbb{I} - \Pi_h)\phi_H\|_{L^2(\Gamma)}$.

Mit der Approximationskonstante C_{apx} aus Korollar 3.2 und der Konstanten C_{inv} der inversen Abschätzung aus Korollar 3.5 gelten

$$\eta_M \leq \widetilde{\eta}_M \leq C_{apx}\sigma(\mathcal{T}_h)^{1/2}\widetilde{\mu}_M \quad \textit{und} \quad \widetilde{\mu}_M \leq \mu_M \leq C_{inv}2^{\ell/2}\eta_M. \tag{3.12}$$

Insbesondere sind die eingeführten Schätzer äquivalent zu η_M. Die Zuverlässigkeit von μ_M und $\widetilde{\mu}_M$ sowie die Effizienz von $\widetilde{\eta}_M$ hängen von der Formregularität $\sigma(\mathcal{T}_h)$ des Netzes ab.

Beweis. ***1. Schritt.*** *$\eta_M \leq \widetilde{\eta}_M$:*
Aus der Bestapproximationseigenschaft der Galerkinprojektion folgt unmittelbar

$$\eta_M = |\!|\!|\phi_H - \mathbb{G}_h\phi_H|\!|\!| \leq |\!|\!|\phi_H - \Pi_h\phi_H|\!|\!| = \widetilde{\eta}_M.$$

2. Schritt. *$\widetilde{\eta}_M \leq C_{apx}\sigma(\mathcal{T}_h)^{1/2}\widetilde{\mu}_M$:*
Mit der Approximationseigenschaft der L^2-Projektion (3.4) erhält man

$$\begin{aligned}\widetilde{\eta}_M = |\!|\!|(\mathbb{I} - \Pi_h)\phi_H|\!|\!| \leq C_{\mathrm{apx}}\|h^{1/2}(\mathbb{I} - \Pi_h)\phi_H\|_{L^2(\Gamma)} &= C_{\mathrm{apx}}\sigma(\mathcal{T}_h)\|\varrho^{1/2}(\mathbb{I} - \Pi_h)\phi_H\|_{L^2(\Gamma)} \\ &= C_{\mathrm{apx}}\sigma(\mathcal{T}_h)^{1/2}\widetilde{\mu}_M.\end{aligned}$$

3. Schritt. *$\widetilde{\mu}_M \leq \mu_M$:*
Man beachte, dass Π_h die elementweise Bestapproximation bezüglich der L^2-Norm ist. Somit ist

$$\widetilde{\mu}_M^2 = \sum_{T_j \in \mathcal{T}_H} P_{T_j}\|\phi_H - \Pi_h\phi_H\|^2_{L^2(T_j)} \leq \sum_{T_j \in \mathcal{T}_H} P_{T_j}\|\phi_H - \mathbb{G}_h\phi_H\|^2_{L^2(T_j)} = \mu_M^2.$$

4. Schritt. *$\mu_M \leq C_{inv}2^{\ell/2}\eta_M$:*
Die lokale inverse Abschätzung (3.7) zeigt

$$\|\varrho^{1/2}(\phi_H - \mathbb{G}_h\phi_H)\|_{L^2(\Gamma)} = \underbrace{\|(\varrho/P)^{1/2}\|_{L^\infty(\Gamma)}}_{=2^{\ell/2}}\|P^{1/2}(\phi_H - \mathbb{G}_h\phi_H)\|_{L^2(\Gamma)} \leq C_{\mathrm{inv}}2^{\ell/2}|\!|\!|\phi_H - \mathbb{G}_h\phi_H|\!|\!|.$$

Somit ist die Ungleichungskette vollständig bewiesen. ∎

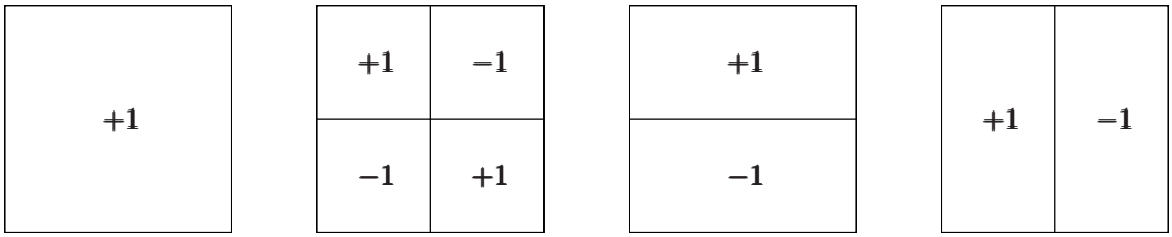


Abbildung 3.1: Für jedes Rechteck $T \in \mathcal{T}_h$ werden $\mathcal{T}_{h/2}$-stückweise konstante Funktionen $\psi_j = \psi_{T,j} \in \mathcal{P}^0(\mathcal{T}_{h/2})$ eingeführt, die auf $\Gamma \backslash T$ durch 0 fortgesetzt werden.

3.4 Vergleich zwischen η_H und η_M

Wir haben zwei verschiedene Arten von Fehlerschätzern eingeführt und eingehend studiert. Der Schätzer η_H zeichnet sich durch minimalen Implementierungsaufwand aus. Allerdings schätzt er nur den Fehler auf einem groben Netz $\mathcal{T}_h$, obwohl, wie bereits angemerkt, numerische Software als Lösung $\phi_{h/2}$ zurückgeben würde. Insofern ist η_H in gewisser Weise suboptimal. Der Glättungsschätzer η_M ist in dieser Hinsicht besser, da er tatsächlich den Fehler auf dem feinen Netz $\mathcal{T}_H$ zuverlässig und effizient approximiert. Allerdings müssen Programmteile, wie etwa $\mathcal{P}^1$-Elemente, implementiert werden. Außerdem ist die Analysis von η_M etwas komplizierter und lässt einige Fragen, wie die geeignete Wahl von ℓ, in Praxis offen.

Man kann aber η_M auch als Fehlerschätzer für $|||\phi - \phi_h|||$ auffassen. Wählen wir außerdem $\ell = 1$, das heißt $\mathcal{T}_H = \mathcal{T}_{h/2}$, so stellt sich heraus, dass η_M und η_H äquivalent sind. Dieses Resultat wird in Erath/Ferraz-Leite/Funken/Praetorius 2007 [12] im zweidimensionalen Fall gezeigt. Wir werden den dreidimensionalen Fall über Rechtecksnetzen untersuchen. Wir erhalten in 3D nur die Aussage $\widetilde{\mu}_H \leq 2\widetilde{\mu}_M$, wo hingegen im zweidimensionalen Fall die Gleichheit gezeigt werden kann. Die Idee, die in Abbildung 3.1 dargestellte Basis für die Berechnung zu benutzen, wurde durch das Studium von Carstensen/Maischak/Stephan 2001 [7] inspiriert.

Satz 3.11. *Der Fehlerschätzer η_M ist bezüglich $|||\phi - \phi_h|||$ stets effizient. Genauer gilt*

$$\eta_M \leq \eta_H. \tag{3.13}$$

Seien $\mathcal{T}_h$ und $\mathcal{T}_{h/2}$ aus Rechtecken bestehenden Paneelierungen, wobei $\mathcal{T}_{h/2}$ durch uniforme Verfeinerung aus $\mathcal{T}_h$ hervorgeht, dann gilt

$$\widetilde{\mu}_H \leq 2\widetilde{\mu}_M. \tag{3.14}$$

Unter der Saturationsannahme (3.8) *ist somit η_M zuverlässig, wobei diese Eigenschaft von der Formregularität $\sigma(\mathcal{T}_h)$ abhängt.*

Beweis. ***1. Schritt.*** *$\eta_M \leq \eta_H$:*
Bezeichne $\mathbb{G}_h^{(1)}$ die Galerkinprojektion auf $\mathcal{P}^1(\mathcal{T}_h)$ und $\mathbb{G}_h$ die Galerkinprojektion auf $\mathcal{P}^0(\mathcal{T}_h)$. Mit der Galerkinorthogonalität gilt für beliebiges $v \in \widetilde{H}^{-1/2}(\Gamma)$ laut Satz von Pythagoras

$$|||v - \mathbb{G}_h^{(1)} v|||^2 + |||\mathbb{G}_h^{(1)} v - \mathbb{G}_h v|||^2 = |||v - \mathbb{G}_h v|||^2.$$

Setzt man für v die diskrete Galerkinlösung $\phi_{h/2}$ ein, so erhält man

$$\eta_M^2 + |||\mathbb{G}_H \phi_{h/2} - \mathbb{G}_h \phi_{h/2}|||^2 = \eta_H^2.$$

2. Schritt. $\widetilde{\mu}_H \leq 2\widetilde{\mu}_M$*:*
Bezeichne $\Pi_h^{(1)}$ die L^2-Projektion auf $\mathcal{P}^1(\mathcal{T}_h)$ und Π_h die L^2-Projektion auf $\mathcal{P}^0(\mathcal{T}_h)$. Ohne Beschränkung der Allgemeinheit sei T das Rechteck $(-1,1)^2$ mit den zugehörigen Rechtecken $T_1,\ldots,T_4$ des Netzes $\mathcal{T}_{h/2}$ in der Numerierung wie in Abbildung 2.2. Laut Satz von Pythagoras gilt

$$\|\phi_{h/2} - \Pi_h^{(1)}\phi_{h/2}\|^2_{L^2(T)} + \|\Pi_h^{(1)}\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)} = \|\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)},$$

bzw. umgeschrieben

$$\|\phi_{h/2} - \Pi_h^{(1)}\phi_{h/2}\|^2_{L^2(T)} = \|\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)} - \|\Pi_h^{(1)}\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)}. \tag{3.15}$$

Wir wollen diese Gleichung nutzen, um die Fehlerschätzer geeignet abzuschätzen. Nehme $\phi_{h/2}$ auf T die Werte a auf T_1, b auf T_2, c auf T_3 und d auf T_4 an. Wir stellen fest, dass die Funktionen $\varphi_1 = 1, \varphi_2 = x, \varphi_3 = y, \varphi_4 = xy$ eine L^2-Orthogonalbasis von $\mathcal{P}^1(T)$ ist. Die in Abbildung 3.1 dargestellten Funktionen $\psi_1,\ldots,\psi_4$ sind eine L^2-Orthogonalbasis von $\mathcal{P}^0(T)$. Daher lassen sich die Terme

$$\begin{aligned}
\|\Pi_h^{(1)}\phi_{h/2} - \Pi_h\phi_{h/2}\|^2 &= \sum_{j=2}^{4} \frac{\langle \phi_{h/2}, \varphi_j\rangle^2}{\|\varphi_j\|^2_{L^2(T)}} \\
\|\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)} &= \sum_{j=2}^{4} \frac{\langle \phi_{h/2}, \psi_j\rangle^2}{\|\psi_j\|^2_{L^2(T)}}
\end{aligned}$$

berechnen. Es gilt

$$\begin{aligned}
\|\psi_j\|^2_{L^2(T)} &= 4, \\
\|\varphi_2\|^2_{L^2(T)} = \|\varphi_3\|^2_{L^2(T)} &= 4/3, \\
\|\varphi_4\|^2_{L^2(T)} &= 4/9.
\end{aligned}$$

Elementar lassen sich die Produkte

$$\begin{aligned}
\langle \phi_{h/2}, \psi_4\rangle^2 &= (a-b-c+d)^2 & \langle \phi_{h/2}, \varphi_2\rangle^2 &= \frac{1}{4}(a-b-c+d)^2 \\
\langle \phi_{h/2}, \psi_3\rangle^2 &= (a+b-c-d)^2 & \langle \phi_{h/2}, \varphi_3\rangle^2 &= \frac{1}{4}(a+b-c-d)^2 \\
\langle \phi_{h/2}, \psi_2\rangle^2 &= (a-b+c-d)^2 & \langle \phi_{h/2}, \varphi_4\rangle^2 &= \frac{1}{16}(a-b+c-d)^2
\end{aligned}$$

berechnen. Somit erhält man

$$\begin{aligned}
\frac{\langle \phi_{h/2}, \varphi_2\rangle^2}{\|\varphi_2\|^2_{L^2(T)}} &= \frac{3}{16}(a-b-c+d)^2 = \frac{3}{4}\frac{\langle \phi_{h/2}, \psi_4\rangle^2}{\|\phi_4\|^2_{L^2(T)}} \\
\frac{\langle \phi_{h/2}, \varphi_3\rangle^2}{\|\varphi_3\|^2_{L^2(T)}} &= \frac{3}{16}(a+b-c-d)^2 = \frac{3}{4}\frac{\langle \phi_{h/2}, \psi_3\rangle^2}{\|\phi_3\|^2_{L^2(T)}}
\end{aligned}$$

sowie

$$\begin{aligned}
\frac{\langle \phi_{h/2}, \varphi_4\rangle^2}{\|\varphi_4\|^2_{L^2(T)}} &= \frac{9}{64}(a-b+c-d)^2 \\
&\leq \frac{3}{16}(a-b+c-d)^2 = \frac{3}{4}\frac{\langle \phi_{h/2}, \psi_2\rangle^2}{\|\phi_2\|^2_{L^2(T)}}.
\end{aligned}$$

Insgesamt haben wir

$$\|\Pi_h^{(1)}\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)} \leq \frac{3}{4}\|\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)} \tag{3.16}$$

gezeigt. Einsetzen von (3.16) in Gleichung (3.4) führt auf die Ungleichung

$$\|\phi_{h/2} - \Pi_h^{(1)}\phi_{h/2}\|^2_{L^2(T)} \geq \frac{1}{4}\|\phi_{h/2} - \Pi_h\phi_{h/2}\|^2_{L^2(T)}.$$

Nach Gewichtung der Terme mit $\varrho_T^{1/2}$ und Summation über alle Pannele folgt

$$\widetilde{\mu}_H^2 = \|\varrho^{1/2}(\phi_{h/2} - \Pi_h\phi_{h/2})\|^2_{L^2(\Gamma)} \leq 4\|\varrho^{1/2}(\phi_{h/2} - \Pi_h^{(1)}\phi_{h/2})\|^2_{L^2(\Gamma)} = 4\widetilde{\mu}_M^2,$$

womit der Satz vollständig bewiesen ist. ■

Die Idee des $h-h/2$-Schätzers lässt sich dahingehend verallgemeinern, dass man analog einen $p-(p+1)$-Schätzer $\eta_P := |\!|\!|\phi_h^{(p+1)} - \phi_h^{(p)}|\!|\!|$ einführen kann, der mit verschiedenem Polynomgrad arbeitet. Diese Idee wird in Erath/Ferraz-Leite/Funken/Praetorius 2007 [12] eingehend studiert, wobei die Analysis völlig analog zum $h - h/2$ Fall verläuft.

Beide Fehlerschätzer η_H und η_M lassen sich auch für die hypersinguläre Integralgleichung formulieren. Die benötigte lokale inverse Abschätzung sowie eine Approximationsaussage für den modifizierten Scott-Zhang Operator finden sich in Ferraz-Leite/Funken/Praetorius 2007 [15], wo der Schätzer η_M auch im hypersingulären Fall studiert wird. In Ferraz-Leite/Praetorius 2007 [16] wird η_H für die hypersinguläre Integralgleichung studiert.

Kapitel 4

Die Programmbibliothek FLBEM

Für die Durchführung numerischer Experimente wurde eine in C++ entwickelte Programmbibliothek FLBEM verwendet. Die FLBEM enthält mehrere Module, die im Folgenden überblicksmäßig erklärt werden. Es wird dabei besonders auf die Datenstrukturen und die Interaktion der Objekte bei der Netzverwaltung eingegangen. Im Hintergrund werden bei der Manipulation der Systemmatrizen LAPACK, BLAS und HLib verwendet. Die Bibliothek FLBEM ist auf achsenparallele, rechteckige Randelemente ausgelegt. Für diese können die Einträge der Systemmatrix wie in Maischak 2000 [23] analytisch berechnet werden, sodass der Einfluss von Datenfehlern durch Quadraturverfahren ausgeschlossen werden kann.

4.1 Einfache Datenstrukturen

Die in den Dateien *geometrics.h* und *geometrics.cpp* definierten Klassen dienen der Manipulation sehr einfacher geometrischer Objekte, die in allen Bereichen der Bibliothek verwendet werden.

Die Datenstruktur *Matrix* dient der Darstellung von Matrizen. Die Methoden sind weitgehend selbst-

```
1  class Matrix
2  {
3    int rdim,cdim;
4    double* data;
5    public:
6      Matrix();
7      Matrix(int n);
8      Matrix(int n,int m);
9      ~Matrix();
10     void set(int i,int j,double value);
11     double get(int i,int j) const;
12     int get_rowdim() const;
13     int get_coldim() const;
14 };
```

erklärend. Die Einträge werden im Datenfeld *data* im Fortran-Format gespeichert. Dieses Feld wurde im C-Stil als Pointer auf double deklariert, obwohl es in C++ weitaus geeignetere Datenstrukturen, etwa Container vom Typ std :: vektor, gibt. Durch das gewählte Datenformat wird volle Kompatibilität zu allen LAPACK und BLAS Routinen gewährleistet.

Um Punkte im dreidimensionalen Raum darzustellen, dient die Klasse *Point3D*. Der Standardkon-

```
class Point3D
{
  double x_koord,y_koord,z_koord;

  public:
    Point3D();
    Point3D(double x,double y,double z);
    ~Point3D();

    double x() const;
    double y() const;
    double z() const;

    void set_x(double x);
    void set_y(double y);
    void set_z(double z);

    double distance(const Point3D& p) const;

    Point3D operator* (double a);
    Point3D operator/ (double a);
};

bool operator==(const Point3D& p, const Point3D& q);
Point3D operator+ (const Point3D& p,const Point3D& q);
Point3D operator- (const Point3D& p,const Point3D& q);

typedef Point3D Vector;

double norm(const Vector&);
void normalize(Vector&);
double scalarproduct(const Vector&,const Vector&);
Vector vectorproduct(const Vector&,const Vector&);

Vector MVM(const Matrix&,const Vector&);
double* MVM(const Matrix& A, double* v);

double* solveGauss(Matrix& A, double* b);
```

struktor erzeugt den Koordinatenursprung. Die Methode *distance* dient zur Ermittlung der euklidischen Distanz zu einem anderen Punkt. Zur Klasse *Point3D* gibt es eine Reihe von Operatoren. Zwei Punkte werden durch == identifiziert, falls die Koordinaten übereinstimmen. Die Multiplikation mit einem Skalar sowie die Addition und die Subtraktion zweier Punkte ist ebenfalls über die kanonischen Operatoren realisiert. Weiters soll darauf hingewiesen werden, dass die Begriffe *Vector* und *Point3D* austauschbar sind. Das ist insofern sinnvoll, als dass sich die Datenstrukturen der zugehörigen mathematischen Objekte nicht unterscheiden. Die Routine *norm* berechnet die euklidische Norm eines *Vector*-Objektes. Mit *normalize* kann aus einem beliebigen *Vector* der entsprechende Einheitsvektor erzeugt werden. Das Skalarprodukt und das Vektorprodukt sind ebenfalls implementiert. Es gibt zwei

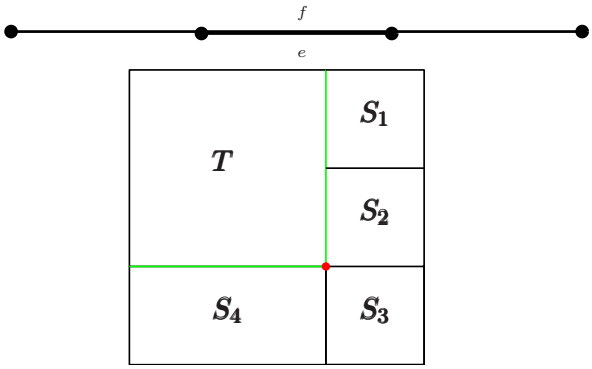


Abbildung 4.1: Oben ist die Kante f eine "Masterkante" von e, weil f eine Obermenge ist. Im unteren Teil des Bildes fasst die Software die Rechtecke T und S_1, S_2 bzw. S_4 als benachbart auf. Die grün markierten Kanten liegen an T an. Obwohl T und S_3 einen gemeinsamen (rot markierten) Punkt besitzen, werden sie von FLBEM nicht als benachbart angesehen.

Versionen der Matrix-Vektor-Multiplikation, die beide mit *MVM* aufgerufen werden. In einem Fall ist eine Manipulation im dreidimensionalen Raum gemeint, im anderen handelt es sich um das Produkt in beliebiger Dimension. Die Funktion *solveGauss* ruft die Routine *dgesv* aus LAPACK auf. Man beachte, dass hierbei insbesondere das Datenfeld der Matrix A überschrieben wird.

Als letzte grundlegende geometrische Klasse wird eine Datenstruktur zur Darstellung der Kante eines geometrischen Objektes deklariert. Im Wesentlichen besteht die Klasse aus den zwei Endpunk-

```
1  class Edge
2  {
3    protected:
4      Point3D p;
5      Point3D q;
6
7    public:
8      Edge();
9      Edge(Point3D point1,Point3D point2);
10     Edge(double x1,double y1,double z1,
11          double x2,double y2,double z2);
12     ~Edge();
13     Point3D p1() const;
14     Point3D p2() const;
15     void set_p1(Point3D point);
16     void set_p2(Point3D point);
17     bool is_masteredge_of(Edge e) const;
18 };
```

ten. Diese werden *protected* deklariert, da *Edge* eine Basisklasse ist und die abgeleiteten Klassen schnellen Zugriff auf diese Daten benötigen. Die interessanteste Methode ist *is_masteredge_of*. Diese ermittelt, ob die als Parameter übergebene Kante e eine Teilmenge der Kante selbst ist. Randelemente werden aus solchen Kanten bestehen. Um festzustellen, ob zwei Randelemente benachbart sind, wird *is_masteredge_of* aufgerufen werden. Da fastreguläre Netze erlaubt sind, kann es hängende Knoten geben. Zwei Randelemente T_1, T_2 sind benachbart, falls eine der Kanten von T_1 eine Obermenge einer der Kanten von T_2 ist oder vice versa. Elemente mit nur einem gemeinsamen Knoten werden vom

Standpunkt der Software aus als nicht benachbart gesehen. Diese Sachverhalte sind in Abbildung 4.1 visualisiert.

4.2 Triangulierung

Beim Design der Triangulierung stand im Vordergrund, eine möglichst flexible Datenstruktur zu erstellen, die das Arbeiten mit unstetigen Ansatzfunktionen über fastregulären Netzen erlaubt. Die Paneelierung sollte sowohl aus ebenen Dreiecken, als auch aus ebenen Rechtecken bestehen können. Ein Mischen verschiedener geometrischer Objekte in einem Netz ist nicht vorgesehen. Obwohl die Implementierung Dreiecke und Rechtecke ermöglicht, wollen wir uns auf den Fall von rechteckigen Paneelen beschränken, da diese auch anisotrop verfeinert werden können. In den numerischen Experimenten in Kapitel 5 werden stets Rechteckspaneelierungen verwendet.

Es wird im Sinn einer besseren Didaktik bei der Beschreibung eine top-down Methode gewählt. Das heißt es wird zunächst die Datenstruktur *Triangulation3D* erklärt. Die Klasse zur Darstellung der

```
1  class Triangulation3D : public std::vector<Shape*>
2  {
3    bool consider_neighbourhood;
4    public:
5      void* data;
6      Triangulation3D();
7      Triangulation3D(std::vector<Shape*>);
8      Triangulation3D(std::vector<Shape*>,bool set_neighbours);
9      Triangulation3D(const Triangulation3D&);
10     ~Triangulation3D();
11     Triangulation3D& operator=(const Triangulation3D&);
12     bool is_considering_neighbourhood() const;
13     void regulate_diameters(double k);
14     void refine();
15     void refine_uniformily();
16 };
```

Paneelierung ist ein Container, der Pointer auf Objekte vom Typ *Shape* zur Darstellung von Randelementen enthält. Bei der Konstruktion des Netzes soll ein initialer Vektor von Pointern übergeben werden. Außerdem kann bestimmt werden, ob das Netz eine Geometrie besitzen soll. Mit einem Netz ohne Geometrie ist im Sinne der Software gemeint, dass Rechtecke keine Information über Nachbarschaftsrelationen verwalten. Insbesondere erlauben geometriefreie Netze beliebig viele hängende Knoten an einer Kante. Der Destruktor zerstört alle im Freispeicher befindlichen Objekte vom Typ *Shape* bevor sich der Container selbst freigibt. Die Routine *regulate_diameters* stellt sicher, dass sich die lokalen Netzweiten h_T, ϱ_T benachbarter Elemente nicht um mehr als den Faktor k unterscheiden. So kann aus jedem Netz eine feinere fastreguläre Triangulierung erzeugt werden. Der Zuweisungsoperator implementiert das Deepcopy für Triangulierungen. Es wird also eine volle Kopie aller Paneele erstellt. Um ein Netz zu verfeinern, müssen die Randelemente zunächst zur Teilung markiert werden. Benachbarte Paneele, die gegebenenfalls zusätzlich verfeinert werden müssen, um sicherzustellen, dass nicht mehr als ein hängender Knoten je Kante entstehen kann, werden automatisch geeignet markiert, falls das Netz eine Geometrie besitzt. Der anschließende Aufruf von *refine* führt dazu, dass sich alle markierten Paneele teilen und die neuentstandenen Randelemente dem Netz hinzugefügt werden. Die Methode *refine_uniformily* implementiert eine uniforme Netzverfeinerung. Da wir uns auf rechteckige

Elemente beschränken, genügt es die Klasse *Rectangle* zu studieren. Diese ist eine Realisierung der abstrakten Klasse *Shape*, die eingeführt wurde um Netze wahlweise mit Rechtecken oder Dreiecken zu ermöglichen. Es ist leicht vorstellbar andere Arten von *Shape*-Klassen zu definieren, sodass die FLBEM Verallgemeinerungspotential besitzt. Ein Rechteck besteht aus vier Kanten vom Typ *Sha-*

```
class Rectangle : public virtual Shape
{
  ShapeEdge ab;
  ShapeEdge bc;
  ShapeEdge cd;
  ShapeEdge da;
  bool hanging_allowed;
  bool marked_red;
  std::list<Shape*> neighbours;
  ShapeEdge* find_neighbour_edge(Shape*);
  public:
    Rectangle();
    Rectangle(bool);
    Rectangle(Point3D,Point3D,Point3D,Point3D);
    Rectangle(Point3D,Point3D,Point3D,Point3D,bool);
    Rectangle(const Rectangle&);
    ~Rectangle();
    std::vector<Point3D> get_vertices() const;
    std::vector<ShapeEdge*> get_edges();
    std::vector<Shape*> get_neighbours();
    void add_neighbour(Shape*);
    void remove_neighbour(Shape*);
    void check_neighbourhood(Shape*);
    void mark_red(bool contact_neighbours=true);
    void mark_red_x(bool contact_neighbours=true);
    void mark_red_y(bool contact_neighbours=true);
    bool is_to_split() const;
    bool is_hanging_allowed() const;
    std::vector<Shape*> split(bool consider_neighbourhood=true);
    void allow_hanging();
    void listen_edge(ShapeEdge*);
    double gram_det() const;
    shape_type what_are_you() const;
};
```

peEdge. Das ist eine aus *Edge* abgeleitete Klasse, die als Kante für Paneele dient. Man beachte, dass die Eckpunkte des Rechtecks nicht explizit als Daten gespeichert werden, sondern nur implizit über die Kanten ermittelt werden können. Jedes Rechteck weiß, ob ein hängender Knoten, also ein Knoten eines Nachbarelementes entlang der eigenen Kante, erlaubt ist. Ein Paneel muss auch wissen, ob es augenblicklich zum Teilen markiert ist. Eine Liste von Pointern auf *Shape* Objekte stellt sicher, dass ein Rechteck seine Nachbarn kennt. Die private Routine *find_neighbour_edge* ist eine reine Hilfsmethode und ermittelt zu gegebenem Nachbarelement die zugehörige angrenzende Kante. Es gibt eine Reihe von Konstruktoren sowie Routinen um die Eckpunkte, Kanten und Nachbarn auszulesen. Die Methode *add_neighbour* trägt ein neues Rechteck in die Liste ein, sofern dieses tatsächlich ein Nachbar ist. Natürlich können Nachbarn auch aus der Liste ausgetragen werden. Mit *check_neighbourhood* wird überprüft, ob ein bestimmtes Paneel benachbart ist. Sollte es, zum Beispiel bedingt durch Ver-

feinerungen, nicht mehr benachbart sein wird es ausgetragen, im anderen Fall wird es, sofern nicht schon vorhanden, in die Liste eingetragen. Die für den Anwender wohl wichtigsten Routinen sind *mark_red*, *mark_red_x* und *mark_red_y*. Die erstgenannte markiert alle Kanten eines Elementes zur Teilung, sodass es isotrop verfeinert wird. Die letzten beiden dienen der anisotropen Verfeinerung eines Elementes. Mit *mark_red_x* werden die Kanten *ab* und *cd* markiert, mit *mark_red_y* entsprechend die anderen Kanten. Der Aufruf von *split* bewirkt, dass ein Element sich teilt. Dabei werden im Freispeicher neue Rechtecke erzeugt. Eine Liste der neuen Rechtecke wird als Vektor von Pointern zurückgegeben, sodass die neuen Paneele zum Beispiel in den *Triangulation3D*-Container eingetragen werden können. Die für das Funktionieren des Netzes sehr wichtige Routine *listen_edge* wird von einer Kante eines benachbarten Elementes immer dann aufgerufen, wenn diese Kante sich verändert, also zum Beispiel, wenn sie zum Teilen markiert wird. In diesem Fall überprüft das Rechteck, ob es einen Anlass gibt sich selbst zu verändern. So kann es passieren, dass ein Rechteck sich selbst markieren muss, da sonst mehr als ein hängender Knoten entlang einer Kante entstehen würde. Beim Aufruf einer Markierungsroutine passieren also eigentlich folgende Dinge:

- Das Rechteck markiert die Kanten, die verfeinert werden sollen.
- Alle Elemente, die benachbart zu einer neu markierten Kante sind, werden durch den Aufruf von *listen_edge* darüber in Kenntnis gesetzt, dass sich die entsprechende Kante geändert hat.
- Ein benachrichtigtes Rechteck sucht die passende eigene Kante, und überprüft, ob es selbst weiter geteilt werden muss.

Die Funktion *gram_det* berechnet die Fläche des Rechtecks. Die Antwort auf *what_are_you* ist RECTANGLE. Diese letzte Methode existiert für den Fall, dass Funktionen auf einem hohen Level, wie zum Beispiel Quadraturroutinen, für Dreiecke und Rechtecke unterschiedlich arbeiten.

Als letzte Klasse, die logisch zum Modul der Triangulierung gehört, wird die aus *Edge* abgeleitete Klasse *ShapeEdge* betrachtet.

```
1  class ShapeEdge : public Edge
2  {
3    bool marked_to_split;
4    std::list<Shape*> neighbours;
5    std::list<Point3D> hanging_nodes;
6    public:
7      ShapeEdge();
8      ShapeEdge(Point3D,Point3D);
9      ShapeEdge(double,double,double,
10                double,double,double);
11     ~ShapeEdge();
12     void add_neighbour(Shape*);
13     void remove_neighbour(Shape* neighbour);
14     void add_hanging(Point3D);
15     void remove_hanging(Point3D point);
16     std::list<Point3D> get_hanging_nodes() const;
17     bool is_hanging() const;
18     bool mark_to_split();
19     void unmark_to_split();
20     bool is_to_split() const;
21     bool corresponds_to(const Edge& edge_to_check) const;
22     void send();
23 };
```

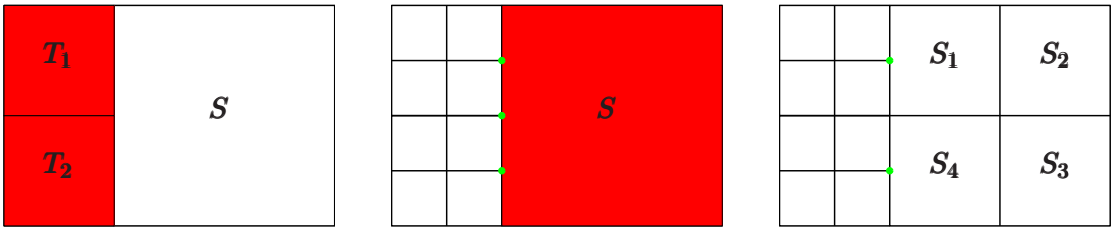


Abbildung 4.2: In der Ausgangskonfiguration werden T_1 und T_2 zum Teilen markiert. Daher muss als Konsequenz auch S geteilt werden, um nicht mehr als einen hängenden Knoten zu erhalten. In einem Zwischenschritt haben sich die Elemente T_1 und T_2 bereits geteilt, dadurch besitzt jetzt S zeitweise drei grün eingezeichnete hängende Knoten und vier Nachbarn an einer Kante. Nach der Teilung von S liegt in der Endkonfiguration nur jeweils ein hängender Knoten an einer Kante von S_1 bzw. S_4 vor.

Eine Kante weiß, ob sie geteilt werden soll, kennt alle benachbarten Rechtecke und kennt alle auf ihr ruhenden hängenden Knoten. Da der Teilungsprozess der Rechtecke einer Paneelierung in einer Schleife erfolgt, kann es, wie in Abbildung 4.2 dargestellt, zeitweise bis zu vier Nachbarn und drei hängende Knoten geben. Die meisten Methoden sind völlig selbsterklärend. Interessant sind die Routinen *corresponds_to*, die ermittelt, ob eine gegeben Kante Obermenge oder Teilmenge ist, und *send*, die die Methode *listen_edge* aller Nachbarn dieser Kante aufruft. Ebenfalls wichtig ist, dass *mark_to_split* den boolschen Wert *false* zurückgibt, falls diese Kante bereits zum Teilen markiert war und sich somit nichts geändert hat.

Es gibt ein eigenes Modul *triangulationio* in dem einige I/O-Routinen für Netze implementiert sind. Zum Beispiel kann die Projektion eines Netzes in die xy-Ebene als Postscriptgrafik gespeichert werden. Die Netzdaten können in einem für Matlab geeigneten Format exportiert werden. Ein vollständiges Listing der Quelltexte inklusive Headerdateien findet sich im Anhang. Als Beispiel für die Verwendung eines Netzes mit FLBEM und zum Testen der Netzverfeinerung wurde das Programm *simulate_mesh_refinement* erstellt, das eine adaptive Netzverfeinerung simuliert.

In Abbildung 4.3 ist der Zeitaufwand und der Speicherverbrauch von *simulate_mesh_refinement* über die Anzahl der generierten Rechtecke bei verschiedenen Simulationen in der jeweils jedes fünfte Element zum Verfeinern markiert wird gezeichnet. Offensichtlich ist der Aufwand linear und die Tatsache, dass eine Triangulierung mit 179005 Rechtecken zur Speicherung nur 65.55 Megabyte benötigt legt nahe, dass bei der Lösung der Symmschen Integralgleichung die Netzverwaltung einen vernachlässigbaren Zusatzaufwand darstellt.

```
1  //*************************************************************//
2  //File:             simulate_mesh_refinement.cpp                //
3  //This file demonstrates the use of the classes and methods defined//
4  //in triangulation.h. It further uses some of the features of the  //
5  //triangulationio.h. The mesh refinement can be simulated with a   //
6  //large variety of parameters. For further information see the     //
7  //documentation file.                                             //
8  //*************************************************************//
9  //Author: Samuel Ferraz-Leite                                     //
10 //Date: 06.2007                                                   //
11 //Version 1.0 stable                                              //
12 //*************************************************************//
13
14 #include<cmath>
15 #include<list>
16 #include<vector>
17 #include<iostream>
18 #include "triangulation.h"
19 #include "triangulationio.h"
20 using namespace triangulation;
21
22 int main()
23 {
24   unsigned int minsize,red_number;
25   bool hanging_allowed,draw;
26   std::cout<<"Simulation␣of␣a␣mesh␣refinement␣in␣2D␣using␣the␣3D␣
        objects\n";
27   std::cout<<"The␣unity-box␣in␣the␣xy-plane␣is␣being␣triangulated\n";
28   std::cout<<"\n";
29   std::cout<<"\nDo␣you␣want␣to␣allow␣hanging␣nodes?␣Type␣0␣for␣no␣and
        ␣1␣for␣yes:␣";
30   std::cin>>hanging_allowed;
31   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
32   Rectangle* R1=new Rectangle(Point3D(0.,0.,0.),Point3D(1.,0.,0.),
33                               Point3D(1.,1.,0.),Point3D(0.,1.,0.),
34                               hanging_allowed);
35   init_triangulation.push_back(R1);
36   Triangulation3D triangulation=Triangulation3D(init_triangulation);
37   std::cout<<"The␣initial␣triangulation␣was␣created.\n";
38   std::cout<<"How␣many␣Shapes␣should␣at␣least␣be␣created:␣";
39   std::cin>>minsize;
40   std::cout<<"How␣many␣Shapes␣should␣be␣refined␣red␣in␣each␣step?\n";
41   std::cout<<"Type␣1␣for␣each␣Shape,␣2␣for␣every␣second,␣etc...:␣";
42   std::cin>>red_number;
43   std::cout<<"Refining␣starts␣now!\n\n";
44   clock_t t1=clock();
45   int counter=0;
46   while(triangulation.size()<minsize)
47   {
48     counter++;
49     Triangulation3D::iterator element=triangulation.begin();
```

```
50      if(red_number>1)
51      {
52        for(unsigned int k=0;k<triangulation.size();k+=red_number)
53        {
54          (*element)->mark_red();
55          unsigned int l=0;
56          while(l<red_number)
57          {
58            ++l;
59            ++element;
60          }
61        }
62        triangulation.refine();
63      }
64      else
65        triangulation.refine_uniformily();
66    }
67    clock_t t2=clock();
68    std::cout<<"The refinemt was completed. "
69             <<triangulation.size()<<" Shapes were\n"
70             <<"created. Since one Shape uses "
71             <<sizeof(Triangle)
72             <<" bytes,\n"
73             <<"this means aproximately "
74             <<triangulation.size()*sizeof(Rectangle)
75             <<" bytes of memory were used.\n\n";
76
77    std::cout<<"The whole process needed "
78             <<double(t2-t1)/CLOCKS_PER_SEC<<" seconds.\n"
79             <<"Time was measured with an accuracy of 1/"
80             <<CLOCKS_PER_SEC<<" seconds.\n"
81             <<counter<<" refinements were computed.\n\n";
82
83    std::cout<<"Do you want to draw the triangulation to a postscript
       file?"
84         " Type 0 for no and 1 for yes: ";
85    std::cin>>draw;
86    if(draw)
87    {
88      std::cout<<"Now starting to draw the triangulation.\n";
89      t1=clock();
90      draw_tr3d_projection_to_xy("simulated_mesh.ps",triangulation);
91      t2=clock();
92      std::cout<<"\nThe drawing took "
93               <<double(t2-t1)/CLOCKS_PER_SEC
94               <<" seconds measuring\n";
95               <<"with the same accuracy as before.\n\n";
96               <<"The drawing was saved to simulated_mesh.ps\n";
97    }
98    std::cout<<"Thank you for using FLBEM\n";
99  }
```

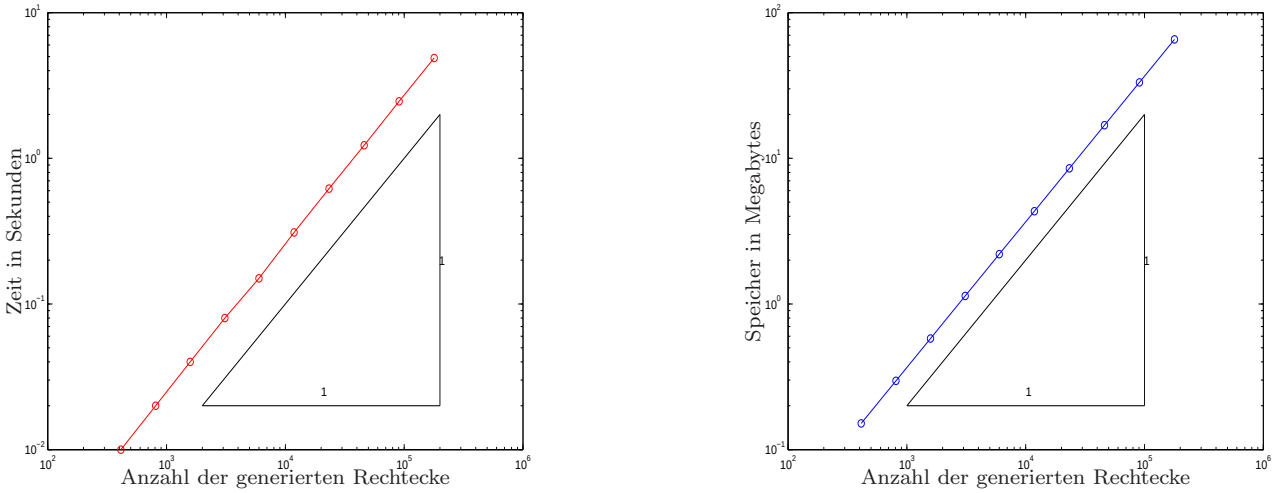


Abbildung 4.3: Zeitaufwand (links) und Speicheraufwand (rechts) der Netzverwaltung. Es sind der Zeitaufwand in Sekunden zur Durchführung von *simulate_mesh_refinement* sowie die dabei anfallende Speicheranforderung in Megabytes über der Zahl der generierten Rechtecke gezeichnet. Das Experiment wurde auf einem Rechner mit zwei AMD Opteron(tm) 248 Prozessoren mit 8GB Arbeitsspeicher und Debian Linux 2.6.17-9 durchgeführt. Der Aufwand ist linear.

4.3 Integraloperatoren

Um mit den Systemmatrizen für die Galerkin-Randelementmethode bequem arbeiten zu können, sind im Modul *integral_operators* alle notwendigen Routinen implementiert. Natürlich ist es sinnvoll, eine abstrakte Klasse einzuführen, die das Aussehen eines Objektes zur Darstellung von Integraloperatoren festlegt. Da bis jetzt aber lediglich das Einfachschichtpotential implementiert ist, wurde einstweilen auf eine solche Klasse verzichtet.

Die Berechnung der Matrixkoeffizienten erfordert Integrations- bzw. Quadraturroutinen, die in *qua-*

```
1  #ifndef BEM_MAIPROGS
2  #define BEM_MAIPROGS
3
4  void liblap3_MP_lapinteg3(double* bx,double* dx1,
5                            double* dx2,double* nx,
6                            int* tx,double* by,
7                            double* dy1,double* dy2,
8                            double* ny,int* ty,int* pvx,
9                            int* pvy,int* pkx,int* pky,
10                           int* pwx,int* pwy,
11                           double* vklmn,double* kklmn,
12                           double* wklmn);
13
14 #endif
```

drature.h und *quadrature.cpp* implementiert sind. Die Stammfunktionen aus Maischak 2000 [23] sind in diesen Dateien rekursiv implementiert. Die Berechnung von $\mathcal{P}^0$-Elementen ist durchaus performant. Allerdings ist die Laufzeit für die Berechnung von Elementen mit Basisfunktionen höheren Polynomgrades aufgrund der rasch steigenden Rekursionstiefe suboptimal. Daher wurde eine Schnittstelle zu den Integrationsroutinen aus dem in Fortran 95 implementierten Programmpaket *maiprogs*

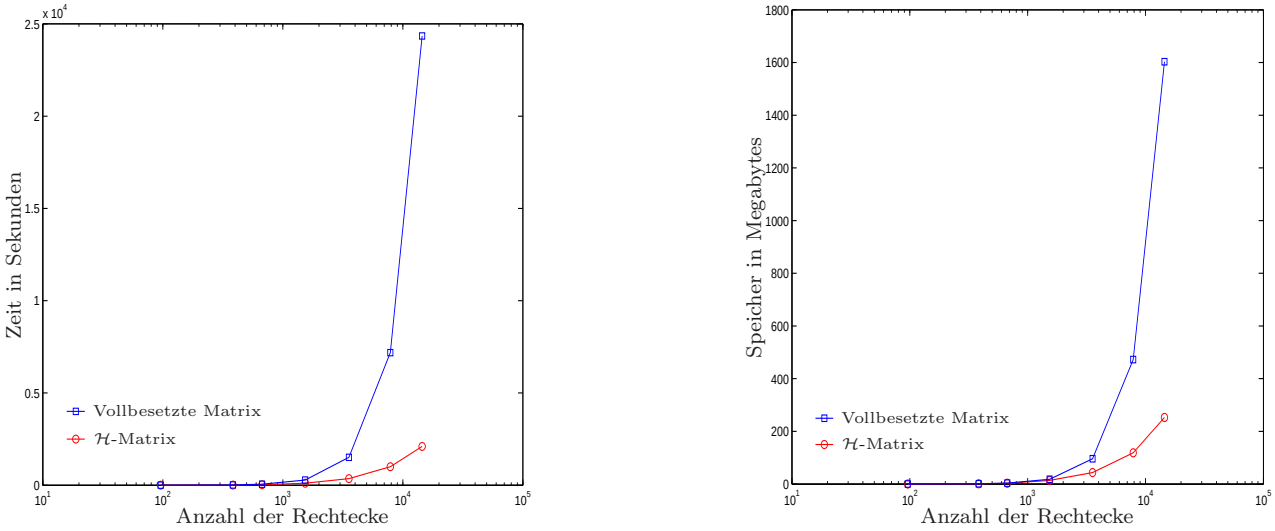


Abbildung 4.4: Zeitaufwand in Sekunden (links) und Speicheraufwand in MB (rechts) beim Aufbau der Systemmatrix für $\mathcal{P}^0$-Elemente über einer durch isotrope Verfeinerungen gewonnenen Netzfolge der Oberfläche des Referenzwürfels $(-1,1)^3$. Die Daten sind für $\mathcal{H}$-Matrizen und vollbesetzte Matrizen dargestellt.

erstellt. In den numerischen Experimenten in Abschnitt 5 wurde zur Berechnung der Einträge in der Steifigkeitsmatrix die Subroutine *lapinteg3* aus dem Modul *liblap3* verwendet. Mit einem Fortran 95 Compiler wurden die Dateien *libgint3.f90*, *libgquad2.f90* und *liblap3.f90* compiliert. Die entstandenen Objektcodes wurden mit der Fortran Standardbibliothek zu einem Archiv zusammengefasst, dass dann beim Linken der Programme, die mit FLBEM erstellt wurden, hinzugefügt wurde. Da Fortran kein call-by-value Prinzip kennt, müssen alle Parameter als Pointer übergeben werden. Die zu C++ kompatible Funktionsdeklaration findet sich in einer eigenen Header-Datei *maiprogs.h*, die mit der Option *extern "C"* eingebunden werden muss. Die dreidimensionalen Vektoren *bx* und *by* enthalten die Koordinaten eines Eckpunktes der beteiligten Rechtecke. Die Variablen *dx1, dx2* bzw. *dy1, dy2* stellen die Richtungsvektoren der Kanten dar. Die Normalvektoren *nx, ny* haben keinen Einfluss auf die Berechnung des Einfachschichtpotentials. Die Integer-Variablen *tx, ty* geben an, ob ein Rechteck, Wert ist 4, oder ein Dreieck, Wert ist 3, vorliegt. In den zweidimensionalen Vektoren *pvx, pvy* wird der Grad der polynomialen Ansatzfunktionen angegeben. Die Ergebnisse der Berechnung sind nach Aufruf der Routine in *vklmn* zu finden.

Die Klasse *SLP* ist die Implementierung des Einfachschichtpotentials V. Es können konstante, lineare sowie eine Mischform aus konstanten und linearen Ansatzfunktionen gewählt werden. Obwohl sich der Anwender nicht mit der konkreten Datenstruktur auseinandersetzen muss, da ein Operator für die Matrix-Vektor-Multiplikation *operator** und ein Operator für das Lösen von linearen Gleichungssystemen *operator/* vorhanden sind, soll hier auf die Möglichkeit eingegangen werden, die Systemmatrix als $\mathcal{H}$-Matrix aufzubauen. Eine genaue Beschreibung von $\mathcal{H}$-Matrizen findet sich in Börm/Grasedyck/Hackbusch 2003 [4]. Durch die Anbindung an die Bibliothek HLib, die am Max-Planck-Institut für Mathematik in den Naturwissenschaften entwickelt wurde und auf www.hlib.org erhältlich ist, kann die Systemmatrix speichereffizient angelegt werden. Um die $\mathcal{H}$-Matrix geeignet approximativ aufzubauen wird das ACA-Verfahren verwendet. In Abbildung 4.6 ist die Struktur einer Systemmatrix abgebildet, die für ein uniformes Netz der Oberfläche des Referenzwürfels $(-1,1)^3$ mit $N = 1536$ Elementen aufgebaut wurde. Es wurde eine Genauigkeit von 10^{-6} gewählt. Die grünen Bereiche sind approximativ gespeichert, die roten voll besetzt. In Tabelle 4.1 ist eine direkte Gegenüberstellung des Speicherverbrauchs und der Rechenzeit für das Anlegen einer vollbesetzten und einer $\mathcal{H}$-Matrix

N	$\widetilde{A}$ [Sek.]	$\widetilde{A}$ [MB]	A [Sek.]	A [MB]	abs. Fehler	rel. Fehler
96	0.69	0.07	1.06	0.07	2.43×10^{-17}	6.60×10^{-17}
384	9.17	1.19	17.48	1.12	1.12×10^{-9}	1.22×10^{-8}
672	25.81	3.32	53.58	3.44	4.49×10^{-9}	5.79×10^{-8}
1536	110.10	13.66	281.86	18.00	1.13×10^{-6}	2.61×10^{-5}
3552	360.26	43.48	1510.34	96.26	1.04×10^{-6}	2.68×10^{-5}
7872	1000.58	118.81	7185.79	472.781	9.30×10^{-7}	2.44×10^{-5}
14496	2105.00	252.974	24344.50	1603.20	1.67×10^{-7}	1.14×10^{-5}

Tabelle 4.1: Die Anzahl der Rechtecke N, eines durch isotrope Verfeinerungen gewonnenen Netzes, das die Oberfläche des Referenzwürfels $(-1,1)^3$ überdeckt, ist in der linken Spalte eingetragen. Dazu sind die Aufbauzeit der zugehörigen $\mathcal{H}$-Matrix $\widetilde{A}$ sowie der entsprechende Speicheraufwand angegeben. Die $\mathcal{H}$-Matrix wurde mittels ACA mit einer Genauigkeit von $10^{-5} \times N^{-1/2}$ mit N der Zahl der Variablen generiert. Auch die zum Aufbau einer vollbesetzten Matrix A verbrauchte Zeit und Speicherkapazität sind zum direkten Vergleich angeführt. Der Speicher wird jeweils in MB angegeben, die Zeit in Sekunden. der absolute Fehler $\|A - \widetilde{A}\|_2$ und der relative Fehler $\|A - \widetilde{A}\|_2 / \|A\|_2$ in der Spektralnorm sind ebenfalls angegeben.

für eine isotrope Netzfolge des Referenzwürfels zu finden. Da beim Feinerwerden des Netzes auch die Approximation der Matrix feiner werden muss, wurde als Genauigkeit für das ACA-Verfahren $10^{-5}N^{-1/2}$ mit N der Zahl der Rechtecke gewählt. Die Berechnungen wurden auf einem Rechner mit zwei AMD Opteron(tm) 248 Prozessoren mit 8GB Arbeitsspeicher und Debian Linux 2.6.17-9 durchgeführt. Wir erkennen für sehr kleine Matrizen über Netzen mit 96 und 348 Rechtecken, dass die $\mathcal{H}$-Matrix sogar mehr Speicher verbraucht, als die vollbesetzte. Das liegt daran, dass Information über die Struktur gespeichert werden muss, sodass ein kleiner Overhead entsteht, der im Vergleich zur geringen Datenmenge noch relativ viel Gewicht hat. Wird das Netz jedoch sukzessive feiner und die Matrizen damit größer, erkennt man die wesentlich bessere Speichereffizienz im Vergleich zum vollbesetzten Fall. Die Laufzeit beim Aufbau der Matrix ist von Beginn an deutlich geringer im Fall der approximativen Abspeicherung. Der große Unterschied liegt aber zum Teil auch daran, dass die FLBEM den Aufbau vollbesetzter Matrizen nicht gänzlich optimiert, es werden nicht alle Symmetrieeigenschaften ausgenutzt. Im Fall der $\mathcal{H}$-Matrix werden weite Teile nur durch Speicherkopien bereits berechneter Abschnitte gefüllt. Wir erkennen, dass der bei der Approximation begangene relative Fehler nach einer anfänglichen Phase, in der die Matrizen noch verhältnissmäßig klein sind, sich in der Größenordnung von 10^{-5} einpendelt und in den letzten drei durchgeführten Rechenschritten schließlich abnimmt. In Abbildung 4.4 sind die Laufzeit und der Speicherverbrauch als Graphen über der Zahl der Rechtecke in einer semilogarithmischen Skala gezeichnet. Man erkennt deutlich, dass der Aufwand im vollbesetzten Fall von höherer Ordnung ist, was mit den Voraussagen der Analysis zu $\mathcal{H}$-Matrizen übereinstimmt, die lediglich einen Aufwand von $\mathcal{O}(N\log N)$ postuliert.

Aufgebaut wird das Objekt vom Typ *SLP*, also die Systemmatrix, mit der Methode *build*, der man im Fall symmetrischer Matrizen nur ein Netz, im Fall unsymmetrischer Matrizen das Netz für die zum Zeilenindex gehörigen Ansatzfunktionen und das Netz für die zum Spaltenindex gehörigen Ansatzfunktionen übergeben muss. Der Parameter *ansatz* kann die Werte *CONSTANT,LINEAR* oder *LINCONST* annehmen. Der optionale Parameter *epsilon* gibt die gewünschte Genauigkeit für den ACA-Algorithmus an. Wird *epsilon*=0 übergeben, so wird die Matrix vollbesetzt aufgebaut.

Zum Lösen linearer Gleichungssysteme wird für approximativ gespeicherte Matrizen das in der HLib implementierte GMRES-Verfahren angewandt. Es wurde nicht die in der HLib vorliegende Implementierung von CG verwendet, da sich diese als fehleranfällig erwies. Für die Verwendung von GMRES wird ein Vorkonditionierer erstellt, der dieselbe Struktur, und damit denselben Speicherverbrauch aufweist wie die Systemmatrix. Es handelt sich um diagonale Vorkonditionierung. Die Methode

```
class SLP{
  function_type ansatz_functions;
  double* reorder(double* v,clustertree* ct);
  double* disorder(double* v,clustertree* ct);
  public:
    ~SLP();
    supermatrix* A;
    clustertree* rct;
    clustertree* cct;
    supermatrix* build_preconditioner();
    void build(Triangulation3D& from_mesh,Triangulation3D& to_mesh,
       function_type ansatz,
              double epsilon=ACA_EPS);
    void build(Triangulation3D& mesh,function_type ansatz,double
       epsilon=ACA_EPS);
    double* operator/ (double* b);
    double* operator* (double* x);
};
```

N	Vorkonditioniert		Nicht vorkonditioniert		Gauss
	Schritte	Zeit	Schritte	Zeit	Zeit
96	3	0.00	3	0.00	0.00
384	23	0.03	23	0.01	0.02
672	107	0.31	57	0.09	0.08
1536	112	1.14	139	1.02	0.81
3552	119	3.55	483	10.93	8.40
7872	121	9.13	1014	57.45	86.81
14496	133	20.93	> 2000	–	532.58

Tabelle 4.2: Anzahl der Schritte und Zeitaufwand für das Lösen eines linearen Gleichungssystems mit N Unbekannten mit $\mathcal{H}$-Matrizen und GMRES mit und ohne diagonaler Vorkonditionierung bzw. mit vollbesetzter Matrix und Gauss-Verfahren.

build_preconditioner ist bewusst für den Anwender zugänglich. Man beachte, dass das Ergebnis vom Typ *supermatrix**, der in der HLib zur Darstellung von $\mathcal{H}$-Matrizen verwendet wird, ist. Sollte eine voll besetzte Matrix vorliegen, wird zum Lösen linearer Gleichungssysteme die Funktion *dgesv* aus LAPACK verwendet, die das Gauss-Verfahren mit Pivotsuche implementiert. Da *dgesv* die übergebene Matrix mit einer permutierten LU-Zerlegung überschreibt, wird zuvor eine volle Kopie erstellt, deren Speicher nach Lösen der Gleichung wieder freigegeben wird. In Tabelle 4.2 sind die Anzahl der Schritte im GMRES-Verfahren mit und ohne Vorkonditionierer sowie der Zeitaufwand für beide Lösungsmethoden für eine isotrope Netzfolge, die die Oberfläche des Referenzwürfels überdeckt, angeführt. Man erkennt, dass für große Matrizen die Kondition zusehends schlechter wird, was an der Adaptivität der Verfeinerung liegt, sodass die Vorkonditionierung den zusätzlichen Speicheraufwand rechtfertigt.

Abschließend sei noch erwähnt, dass die Verwendung von $\mathcal{H}$-Matrizen für anisotrope Netzfolgen zu Problemen führt. Da die Rechtecke immer flacher werden, gewinnen die Rundungsfehler bei der analytischen Berechnung der Einträge der Systemmatrix an Signifikanz. Diese Beobachtung wurde sowohl mit den Routinen der *maiprogs* als auch mit der Implementierung im Modul *quadrature* der FLBEM

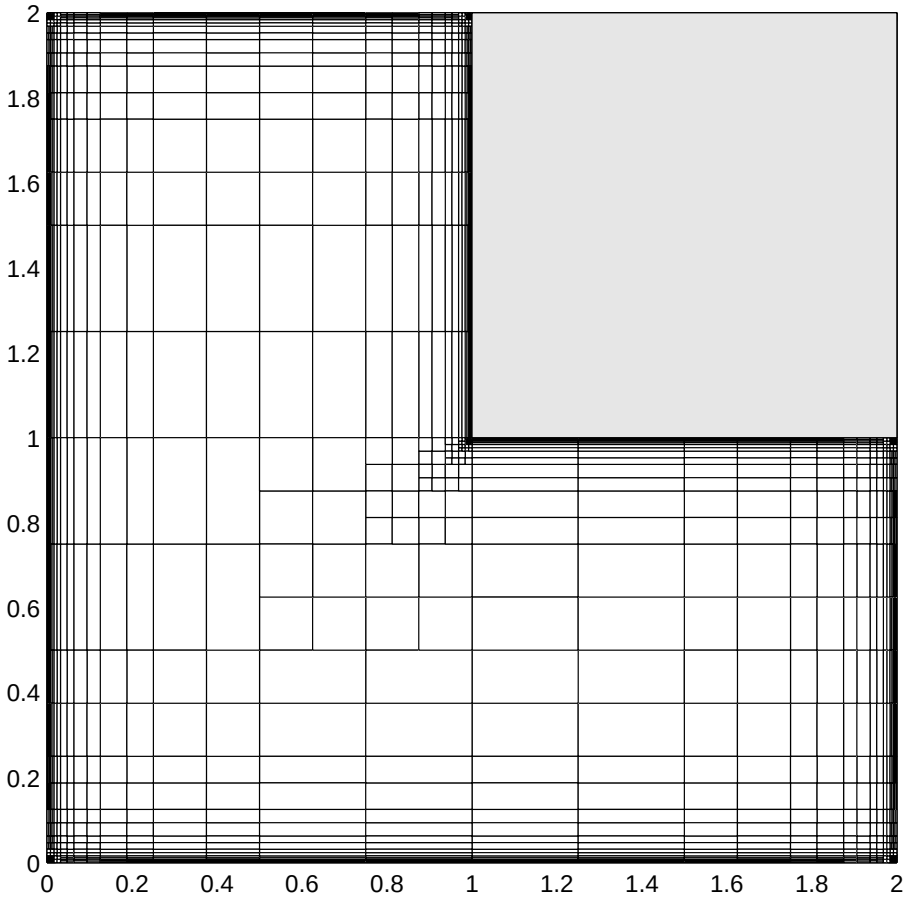


Abbildung 4.5: Bei der Berechnung der $\mathcal{P}^1$-Systemmatrix über diesem Netz ist der Einfluss von Rundungsfehlern so groß, dass die resultierende Matrix nicht positiv definit ist.

gemacht. Insbesondere muss die Genauigkeit für die Approximation durch ACA rasch in der Größenordnung 10^{-14} gewählt werden, um zuverlässige Ergebnisse zu erhalten. Bei einer solchen Genauigkeit ist allerdings die Zeit- und Speicherersparnis nicht relevant, sodass empfohlen wird voll besetzte Matrizen zu verwenden, die den Gesamteffekt von Approximationsfehlern minimieren. Der Einfluss der Rundungsfehler steigt außerdem mit dem Polynomgrad. Für $\mathcal{P}^1$-Elemente und vergleichsweise feine anisotrope Netze wird die Berechnung der Systemmatrix instabil. So ist die über dem in Abbildung 4.5 dargestellten Netz aufgebaute vollbesetzte Matrix weder mit den Integrationsroutinen aus *maiprogs* noch mit denen aus *quadrature* positiv definit, was im Widerspruch zur Analysis steht. Außerdem ist die verwendete Strategie des Panel-Clusterings im anisotropen Fall suboptimal. In Graham/Grasedyck/Hackbusch/Sauter 2006 [18] wird eine bessere Methode vorgestellt.

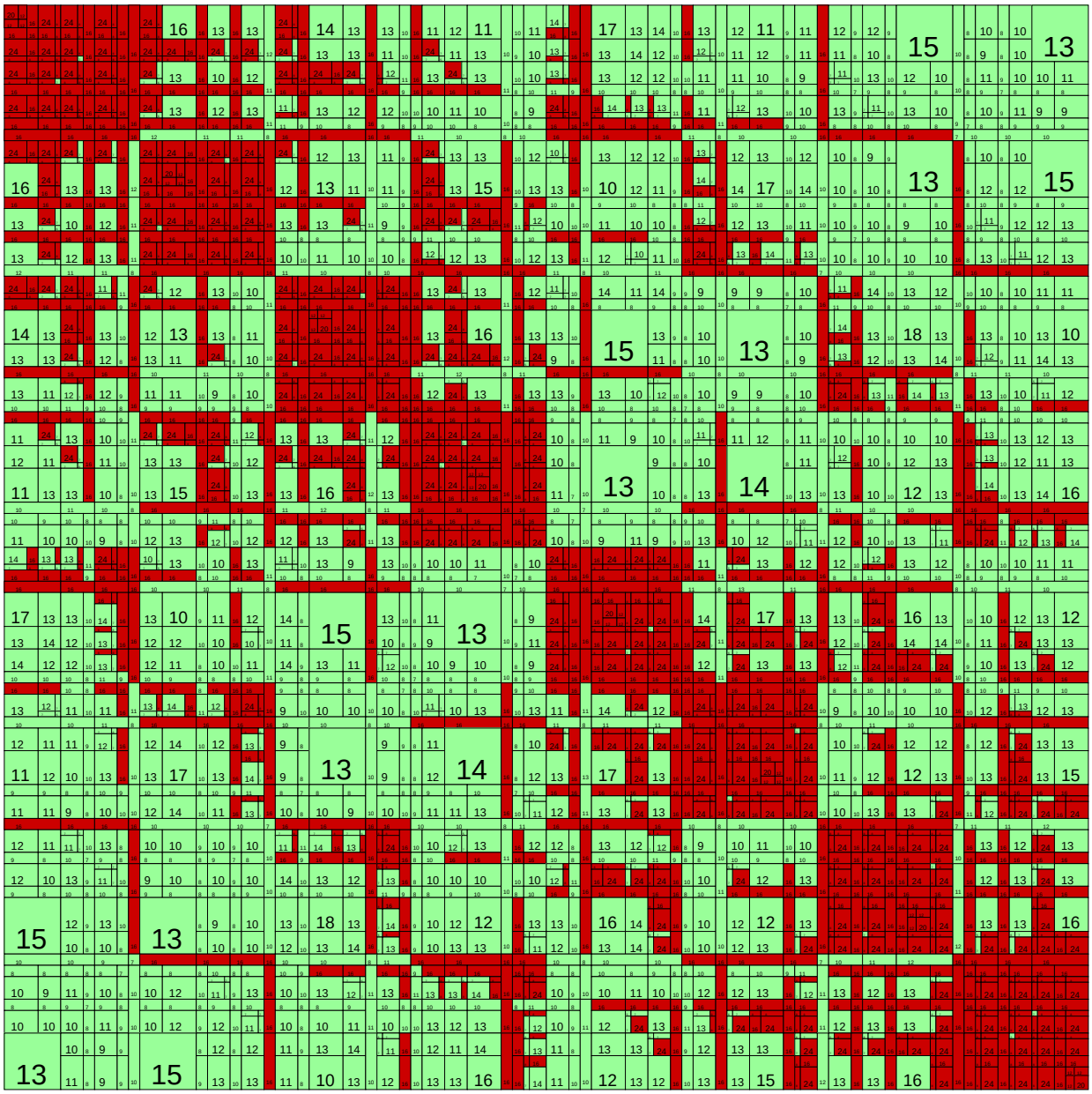


Abbildung 4.6: Exemplarische $\mathcal{H}$-Matrix für ein uniformes Netz, dass mit $N = 1536$ Rechtecken die Oberfläche des Referenzwürfels $(-1, 1)^3$ umspannt. Grüne Bereiche werden approximativ, rote vollbesetzt gespeichert.

Kapitel 5

Numerische Experimente

Wir werden in diesem Abschnitt zunächst einen Algorithmus definieren, der zu einer isotropen adaptiven Netzverfeinerung führt. Mit einer Heuristik wird dieser anschließend wie in Ferraz-Leite/Praetorius 2007 [16] modifiziert, um anisotrope Netzfolgen zu ermöglichen. Die Analysis der in Kapitel 3 eingeführten Fehlerschätzer wird anhand numerischer Experimente überprüft. Da wir uns nur mit der Schätzung des Verfahrensfehlers befasst haben, werden wir uns auf das Lösen der Gleichung

$$V\phi = 1$$

mit konstanter rechter Seite auf verschiedenen Oberflächen beschränken. Durch die Verwendung von FLBEM können die Einträge der Systemmatrix analytisch berechnet werden. Datenfehler durch Quaduraturverfahren oder Interpolation der rechten Seite sind daher im wesentlichen ausgeschlossen, und es kann das Verhalten der Fehlerschätzer unabhängig von äußeren oder inneren unerwünschten Einflüssen beobachtet werden. In diesem Sinne werden die Experimente mit vollbesetzten Matrizen durchgeführt. Der Quelltext der verwendeten Software ist im Anhang zu finden.

Um den Fehler einer Approximation ϕ_h in der Energienorm berechnen zu können, wurde die aus der Galerkinorthogonalität folgende Darstellung

$$|||\phi - \phi_h||| = (|||\phi|||^2 - |||\phi_h|||^2)^{1/2}$$

verwendet. Die unbekannte Energienorm $|||\phi|||$ wird geschätzt, indem wir eine Folge von Approximationen $|||\phi_h|||$ über einer uniformen Netzfolge berechnen. Dabei werden die Systemmatrizen, um mit möglichst vielen Freiheitsgraden rechnen zu können, als $\mathcal{H}$-Matrizen mit einer Genauigkeit von $10^{-6}N^{-1/2}$ mit N der Zahl der Rechtecke aufgebaut. Aus der erhaltenen Folge wird mit der Aitkinschen Δ^2-Methode anschließend ein Wert extrapoliert, der sich im numerischen Experiment als hinreichend exakt erwiesen hat.

Alle Experimente wurden auf einem Rechner mit zwei AMD Opteron(tm) 248 Prozessoren mit jeweils 2.2 GHz durchgeführt, es standen 8GB Arbeitsspeicher zur Verfügung. Das zugrundeliegende Betriebssystem war Debian Linux 2.6.17-2-amd64. Zum Kompilieren der Bibliothek FLBEM sowie der verwendeten Programme kam gcc in der Version 4.1.2 (20060901) zum Einsatz.

5.1 Adaptive Algorithmen

5.1.1 Adaptiver isotroper Algorithmus

Im Folgenden bezeichnet η einen der Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \eta_M$ und $\widetilde{\eta}_M$. Analog bezeichnet μ einen entsprechend lokalisierten Schätzer mit den zugehörigen Verfeinerungsindikatoren μ_j. Um zu gewährleisten, dass eine Folge von Netzen isotrop ist, werden zum Verfeinern markierte Rechtecke geteilt,

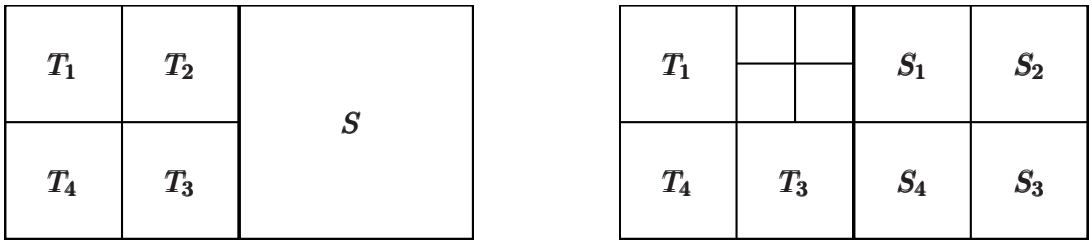


Abbildung 5.1: Ein hängender Knoten je Kante ist erlaubt (links). Falls in der linken Anordnung das Element T_2 zum Verfeinern markiert worden ist, wird auch S markiert. Es ergibt sich nach Teilung die rechts abgebildete Anordnung.

indem alle Seiten halbiert werden. So entstehen vier neue gleichartige Rechtecke wie in Abbildung 2.2 dargestellt. Bei dieser Vorgehensweise bleibt offensichtlich die Formregularität h/ϱ auf jedem Element, und somit auf dem gesamten Netz, erhalten.

Algorithmus 5.1. *Sei $\varepsilon > 0$ eine gegebene Toleranz und $\theta \in [0,1]$ der Adaptivitätsparameter. Wir setzen zu Beginn $\ell = 0$ und fangen mit einem Startnetz $\mathcal{T}_h^{(0)}$ an.*

(i) Verfeinere $\mathcal{T}_h^{(\ell)}$ uniform, um $T_{h/2}^{(\ell)}$ zu erhalten.

(ii) Berechne die Galerkinlösung $\phi_{h/2}^{(\ell)}$.

(iii) Berechne den Fehlerschätzer η und brich den Algorithmus ab, falls $\eta < \varepsilon$.

(iv) Berechne die Verfeinerungsindikatoren μ_j.

(v) Markiere ein Element $T_j \in \mathcal{T}_h^{(\ell)}$, falls $\mu_j \geq \theta \max\{\mu_k | k = 1, \ldots, n\}$.

(vi) Verfeinere die markierten Elemente, um $T_h^{(\ell+1)}$ zu erhalten. Setze $\ell \mapsto \ell + 1$ und gehe zu Schritt (i).

Man beachte, dass sich die Fehlerschätzer η_M und η_H dahingehend unterscheiden, dass η_M den Fehler $|||\phi - \phi_{h/2}|||$ auf dem feinen Netz schätzt, hingegen η_H den Verfahrensfehler $|||\phi - \phi_h|||$ auf dem groben Netz approximiert. Da es jedoch nicht immer möglich ist, durch Vergröberungen eines gegebenen Netzes $\mathcal{T}_{h/2}$ ein geeignetes Netz $\mathcal{T}_h$ zu konstruieren, wird μ_M als Verfeinerungsindikator für $\mathcal{T}_h$ verwendet. Bei der Visualisierung der numerischen Ergebnisse wird auf diesen Unterschied insofern Rücksicht genommen, als dass die zu η_H gehörigen Daten in Bezug auf die Zahl der Elemente in $\mathcal{T}_h$ und die zu η_M gehörigen Daten in Bezug auf die Zahl der Elemente in $\mathcal{T}_{h/2}$ abgebildet werden.

Bemerkung. In Schritt (vi) von Algorithmus 5.1 müssen gegebenenfalls zusätzliche Elemente verfeinert werden, um die Beschränktheit der Zahl der hängenden Knoten und somit die K-Netzeigenschaft zu gewährleisten. Im numerischen Experiment wurde die Zahl der hängenden Knoten stets durch FL-BEM auf 1 eingeschränkt, sodass sich zusätzliche Verfeinerungen wie in Abbildung 5.1 ergeben. Mit der Wahl $\theta = 0$ erhält man eine uniforme Netzfolge. Es wurde $\theta = 0.5$ für die Adaptivität gewählt.

5.1.2 Anisotrope Strategie

Wie wir im Folgenden sehen werden, ist eine isotrope Verfeinerungsstrategie nicht hinreichend, um die optimale Konvergenzrate $\mathcal{O}(N^{-3/4})$ zu erhalten. Um Kantensingularitäten effektiv zu approximieren, müssen lange, dünne Rechtecke zugelassen werden.

Schritt (v) von Algorithmus 5.1 wird durch eine Heuristik zur Steuerung anisotroper Verfeinerung erweitert. Das heuristische Kriterium soll entscheiden, ob ein markiertes Element $T \in \mathcal{T}_h$ in zwei oder vier Elemente, wie in Abbildung 5.2 dargestellt, geteilt werden soll. Für jeden der vorgestellten Fehlerschätzer muss die diskrete Lösung $\phi_{h/2}$ berechnet werden. Sie darf also als bekannt vorausgesetzt werden. Seien $T_1, \dots, T_4$ die zu T gehörenden Elemente des Netzes $\mathcal{T}_{h/2}$, wobei dieselbe Nummerierung wie in Abbildung 2.2 eingehalten wird. Wir betrachten erneut die vier stückweise konstanten Funktionen $\psi_{T,j} \in \mathcal{P}^0(\mathcal{T}_{h/2})$, die in Abbildung 3.1 dargestellt sind. Die Menge $\{\psi_{T,1}, \dots, \psi_{T,4}\}$ bildet eine L^2-Orthogonalbasis von $\mathcal{P}^0(\{T_1, \dots, T_4\})$. Daher kann $\phi_{h/2}|_T \in \mathcal{P}^0(\{T_1, \dots, T_4\})$ als

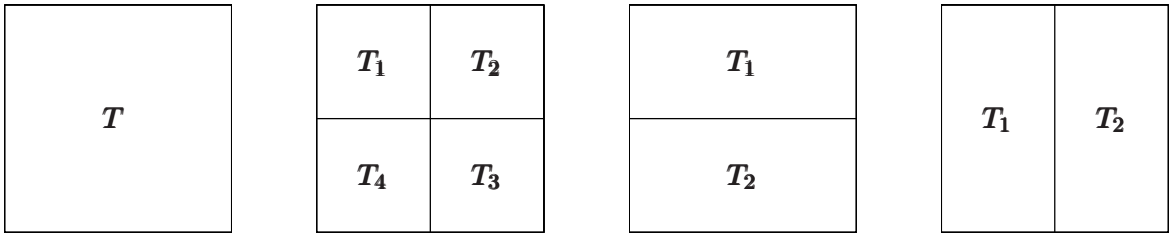
$$\phi_{h/2}|_T = \sum_{j=1}^{4} c_{T,j}\psi_{T,j}$$

mit den Fourierkoeffizienten

$$c_{T,j} = \frac{(\psi_{T,j}, \phi_{h/2})_{L^2(\Gamma)}}{\|\psi_{T,j}\|^2_{L^2(\Gamma)}}$$

dargestellt werden. Um zu entscheiden welche Art der Verfeinerung am geeignetsten erscheint, wird folgendermaßen vorgegangen:

- Falls $c_{T,3}$ signifikant größer als die Fourierkoeffizienten $c_{T,2}$ und $c_{T,4}$ ist, verläuft die diskrete Lösung $\phi_{h/2}$ vergleichsweise konstant in horizontaler Richtung. Daher ist eine vertikale Verfeinerung wie in Abbildung 5.2 scheinbar effizienter als eine isotrope Verfeinerung.
- Falls $c_{T,4}$ signifikant größer als die Fourierkoeffizienten $c_{T,2}$ und $c_{T,3}$ ist, verläuft die diskrete Lösung $\phi_{h/2}$ vergleichsweise konstant in vertikale Richtung. Daher ist eine horizontale Verfeinerung wie in Abbildung 5.2 scheinbar effizienter als eine isotrope Verfeinerung.
- Sollte keiner der zwei bereits genannten Fälle eintreten, wird das Element isotrop verfeinert.



markiertes Element | isotrope Verfeinerung | vertikale Verfeinerung | horizontale Verfeinerung

Abbildung 5.2: Die Modifikation von Algorithmus 5.1 liefert ein Kriterium, ob ein markiertes Rechteck $T \in \mathcal{T}_h$ (links) isotrop zu vier Elementen $T_1, \dots, T_4$ oder anisotrop zu zwei Elementen T_1 und T_2 verfeinert werden soll. In letzterem Fall entscheidet der Algorithmus, ob eine vertikale oder eine horizontale Verfeinerung sinnvoller ist.

Unter signifikant größer verstehen wir, dass für einen Anisotropieparameter $\tau \in [0, 1]$ zum Beispiel $c_{T,3}$ als signifikant größer als $c_{T,2}$ und $c_{T,4}$ erachtet wird, falls

$$\tau|c_{T,3}| \geq (|c_{T,2}|^2 + |c_{T,4}|^2)^{1/2}.$$

uniform		isotrop		anisotrop	
N	q	N	q	N	q
24	0.6457	24	0.6457	24	0.6457
96	0.6523	96	0.6523	96	0.6523
384	0.6437	268	0.6508	168	0.6508
1536	0.6392	384	0.6403	288	0.6406
6144	0.6378	888	0.6335	384	0.6358
		1968	0.6267	504	0.6353
		3624	0.6191	720	0.6257
		4416	0.6229	864	0.6206
		6150	0.6183	1128	0.5997
				1512	0.5777
				1728	0.5804
				2280	0.5666
				2688	0.5695
				3624	0.5929
				4512	0.6354
				6024	0.5696

Tabelle 5.1: Experimentelle Saturationskonstante $q = \frac{\|\phi-\phi_{h/2}\|}{\|\phi-\phi_h\|}$ im Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzverfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

Für die numerischen Experimente wurde, um Anisotropie zu erhalten, $\tau = 0.5$ gewählt. Es soll noch darauf eingegangen werden, dass die Fourierkoeffizienten $c_{T,j}$ einfach zu berechnen sind und keinen bedeutenden Mehraufwand in der Implementierung verursachen. Als Basis für $\mathcal{P}^0(\mathcal{T}_{h/2})$ wird die Menge der charakteristischen Funktionen χ_T gewählt. Insbesondere liegt $\phi_{h/2}$ in der Darstellung

$$\phi_{h/2}|_T = \sum_{j=1}^{4} \lambda_{T,j}\chi_{T,j}$$

vor. Wegen

$$\|\psi_{T,j}\|^2_{L^2(T)} = |T| \quad \text{und} \quad (\psi_{T,j}, \phi_{h/2})_{L^2(T)} = \sum_{k=1}^{4} \lambda_{T,k} \int_{T_k} \psi_{T,j} dx,$$

wobei das Integral die Werte $\pm|T|/4$ annimmt, lassen sich die Fourierkoeffizienten durch

$$\begin{pmatrix} c_{T,1} \\ c_{T,2} \\ c_{T,3} \\ c_{T,4} \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \lambda_{T,1} \\ \lambda_{T,2} \\ \lambda_{T,3} \\ \lambda_{T,4} \end{pmatrix}$$

berechnen.

5.2 Kapazität des Referenzwürfels $(-1, 1)^3$

Als erstes Beispiel wollen wir die Symmsche Integralgleichung

$$V\phi = 1$$

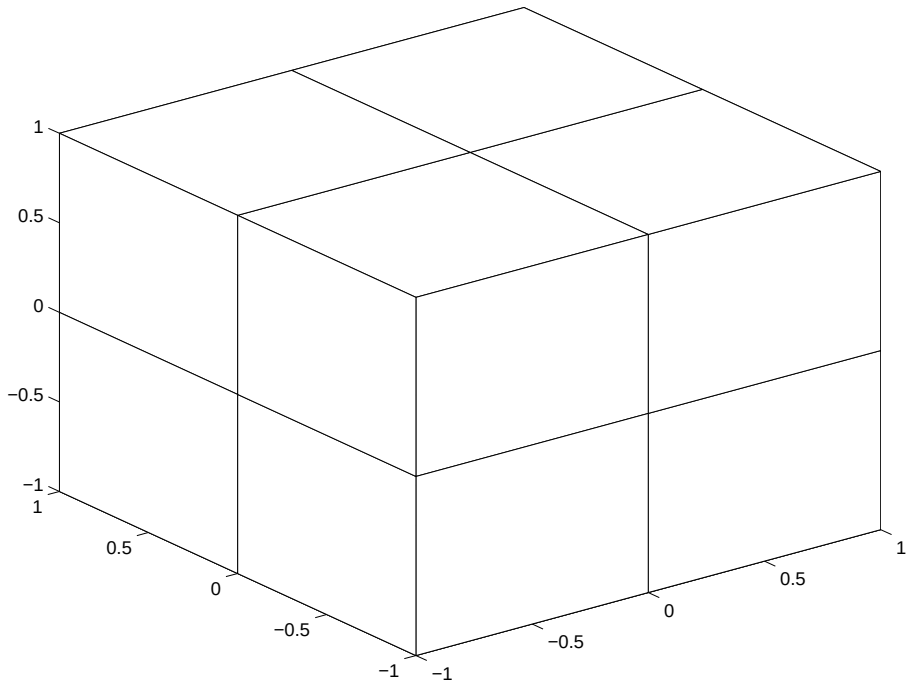


Abbildung 5.3: Startnetz $\mathcal{T}_h^{(0)}$ mit $N = 24$ Elementen für das Kapazitätsproblem 5.2.

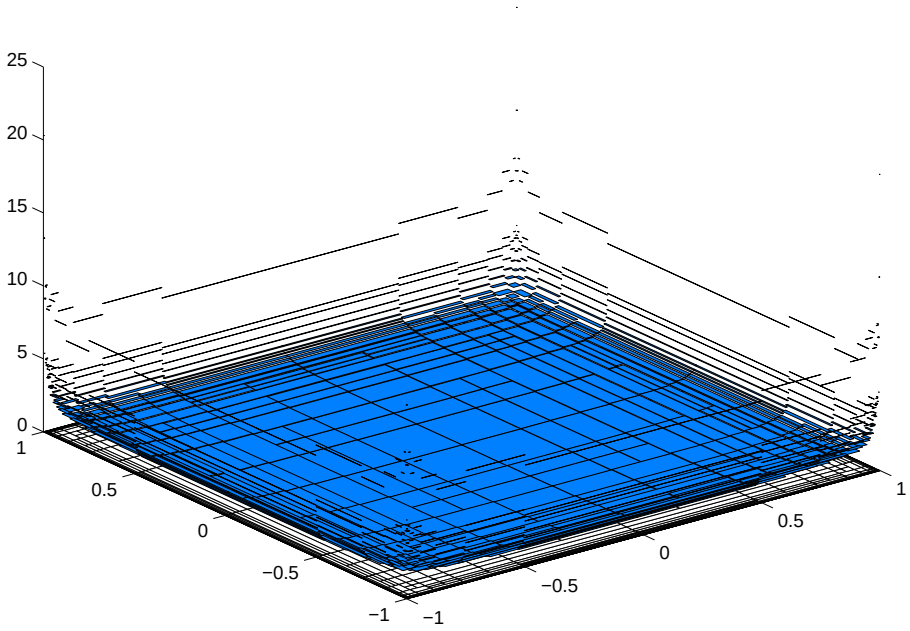


Abbildung 5.4: Teil der diskreten Lösung $\phi_h^{(14)}$ über einer Seite der Triangulierung $\mathcal{T}_h^{(14)}$ mit $N = 4512$ Elementen in Kapazitätsproblem 5.2, die mit $\widetilde{\mu}_H$ als Indikator und anisotroper Verfeinerung generiert wurde. Die Lösung weist Singularitäten entlang der Kanten auf.

uniform			isotrop			anisotrop		
N	$f(q)$	C_{rel}	N	$f(q)$	C_{rel}	N	$f(q)$	C_{rel}
24	1.310	1.310	24	1.310	1.310	24	1.310	1.310
96	1.319	1.319	96	1.319	1.319	96	1.319	1.319
384	1.307	1.307	268	1.317	1.317	168	1.317	1.317
1536	1.300	1.300	384	1.302	1.303	288	1.302	1.303
6144	1.298	1.298	888	1.292	1.295	384	1.296	1.298
			1968	1.283	1.289	504	1.295	1.298
			3624	1.273	1.284	720	1.282	1.288
			4416	1.278	1.283	864	1.275	1.284
			6150	1.272	1.283	1128	1.250	1.259
						1512	1.225	1.231
						1728	1.228	1.236
						2280	1.214	1.233
						2688	1.217	1.235
						3624	1.242	1.281
						4512	1.295	1.313
						6024	1.217	1.364

Tabelle 5.2: Experimentelle Zuverlässigkeitskonstante C_{rel} und $f(q) = 1/\sqrt{1-q^2}$ für die experimentelle Saturationskonstante q im Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzfolgen. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

N	$\|\!\|\!\|\phi - \phi_h\|\!\|\!\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
24	0.546	–	0.417	0.419	1.094	1.094
96	0.353	0.315	0.267	0.284	0.801	0.783
384	0.230	0.308	0.175	0.186	0.534	0.520
1536	0.148	0.317	0.114	0.121	0.347	0.337
6144	0.095	0.323	0.073	0.077	0.222	0.216

Tabelle 5.3: Fehler $|\!|\!|\phi - \phi_h|\!|\!|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine uniforme Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

auf der Oberfläche Γ des Referenzwürfels $(-1,1)^3$ lösen. Extrapolation mit der Aitkinschen Δ^2-Methode führt zu einer geschätzten Energienorm $|\!|\!|\phi|\!|\!|^2 = 16.60470319300621$ der unbekannten Lösung ϕ. Das grobe Startgitter $\mathcal{T}_h^{(0)}$ mit $N = 24$ Rechtecken ist in Abbildung 5.3 zu sehen. Um einen Eindruck des Verhaltens der unbekannten Lösung ϕ zu bekommen, ist in Abbildung 5.4 die diskrete Funktion $\phi_h^{(14)}$ über einer Seite des zugehörigen durch $\widetilde{\mu}_H$-adaptive anisotrope Verfeinerung gewonnenen Netzes $\mathcal{T}_h^{(14)}$ abgebildet. Die Funktion verhält sich auf allen Flächen des Würfels gleich. Es sind Singularitäten an allen Ecken und entlang der Kanten zu erkennen.

In der Theorie haben wir für die Zuverlässigkeit von η_H die Saturationsannahme vorausgesetzt. In Tabelle 5.1 sind die experimentellen Größen

$$q = \frac{|\!|\!|\phi - \phi_{h/2}|\!|\!|}{|\!|\!|\phi - \phi_h|\!|\!|}$$

für uniforme, isotrope und anisotrope Netzfolgen, die sich aus $\widetilde{\mu}_H$-adaptiver Verfeinerung ergeben festgehalten. Mit N wird die Zahl der Rechtecke des groben Netzes bezeichnet. In Abbildung 5.5 sind die Werte als Graphen abgebildet. Man erkennt sofort, dass q nach oben durch 0.66 beschränkt bleibt, sodass die Saturationsannahme experimentell bestätigt wird.

N	$\|\|\|\phi - \phi_h\|\|\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
24	0.5460	–	0.417	0.419	1.094	1.094
96	0.3526	0.316	0.267	0.284	0.801	0.783
168	0.2813	0.403	0.214	0.231	0.675	0.650
384	0.2022	0.400	0.155	0.166	0.482	0.466
888	0.1373	0.462	0.106	0.113	0.327	0.317
1968	0.0912	0.514	0.071	0.075	0.219	0.213
3624	0.0678	0.485	0.053	0.056	0.165	0.161
4416	0.0590	0.703	0.046	0.049	0.143	0.140
6150	0.0507	0.458	0.040	0.042	0.124	0.121

Tabelle 5.4: Fehler $\|\|\|\phi - \phi_h\|\|\|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_H$-adaptive isotrope Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

N	$\|\|\|\phi - \phi_h\|\|\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
24	5.46×10^{-1}	–	4.16×10^{-1}	4.19×10^{-1}	1.09×10^{0}	1.09×10^{0}
96	3.53×10^{-1}	0.316	2.67×10^{-1}	2.84×10^{-1}	8.02×1^{-1}	7.83×10^{-1}
168	2.81×10^{-1}	0.403	2.14×10^{-1}	2.31×10^{-1}	6.75×10^{-1}	6.49×10^{-1}
288	2.02×10^{-1}	0.611	1.56×10^{-1}	1.66×10^{-1}	4.81×10^{-1}	4.66×10^{-1}
384	1.49×10^{-1}	1.061	1.15×10^{-1}	1.22×10^{-1}	3.48×10^{-1}	3.40×10^{-1}
504	1.18×10^{-1}	0.865	9.11×10^{-2}	9.71×10^{-2}	2.83×10^{-1}	2.74×10^{-1}
720	8.05×10^{-2}	1.071	6.28×10^{-2}	6.68×10^{-2}	1.93×10^{-1}	1.88×10^{-1}
864	6.08×10^{-2}	1.542	4.76×10^{-2}	5.10×10^{-2}	1.48×10^{-1}	1.44×10^{-1}
1128	4.41×10^{-2}	1.203	3.53×10^{-2}	3.79×10^{-2}	1.08×10^{-1}	1.05×10^{-1}
1512	3.27×10^{-2}	1.016	2.67×10^{-2}	2.89×10^{-2}	8.08×10^{-2}	7.92×10^{-1}
1728	2.90×10^{-2}	0.916	2.36×10^{-2}	2.55×10^{-2}	6.81×10^{-2}	6.66×10^{-2}
2280	2.29×10^{-2}	0.852	1.88×10^{-2}	2.04×10^{-2}	5.02×10^{-2}	4.92×10^{-2}
2688	2.06×10^{-2}	0.627	1.70×10^{-2}	1.83×10^{-2}	4.25×10^{-2}	4.15×10^{-2}
3624	1.67×10^{-2}	0.727	1.35×10^{-2}	1.46×10^{-2}	3.18×10^{-2}	3.12×10^{-2}
4512	1.43×10^{-2}	0.531	1.10×10^{-2}	1.25×10^{-2}	2.61×10^{-2}	2.53×10^{-2}

Tabelle 5.5: Fehler $\|\|\|\phi - \phi_h\|\|\|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_H$-adaptive anisotrope Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

In Kapitel 3 haben wir

$$\|\|\|\phi - \phi_h\|\|\| \leq C_{\mathrm{rel}} \eta_H$$

mit der Konstante

$$C_{\mathrm{rel}} = \frac{1}{\sqrt{1 - q^2}}$$

gezeigt, wobei q die Saturationskonstante bezeichnet. In Tabelle 5.2 sind die experimentelle Zuverlässigkeitskonstante

$$C_{\mathrm{rel}} = \frac{\|\|\|\phi - \phi_h\|\|\|}{\eta_H}$$

sowie die Werte der Funktion $f(q) = 1/\sqrt{1 - q^2}$ für die experimentelle Saturationskonstante über der Zahl der Elemente im groben Gitter angeführt. Offensichtlich sollte mit den experimentellen Werten $C_{\mathrm{rel}} = f(q)$ gelten. Wir beobachten allerdings in den adaptiven Fällen jeweils kleinere Abweichungen. Diese können etwa dadurch entstehen, dass bei der Extrapolation der Energienorm $\|\|\|\phi\|\|\|$ der exakten

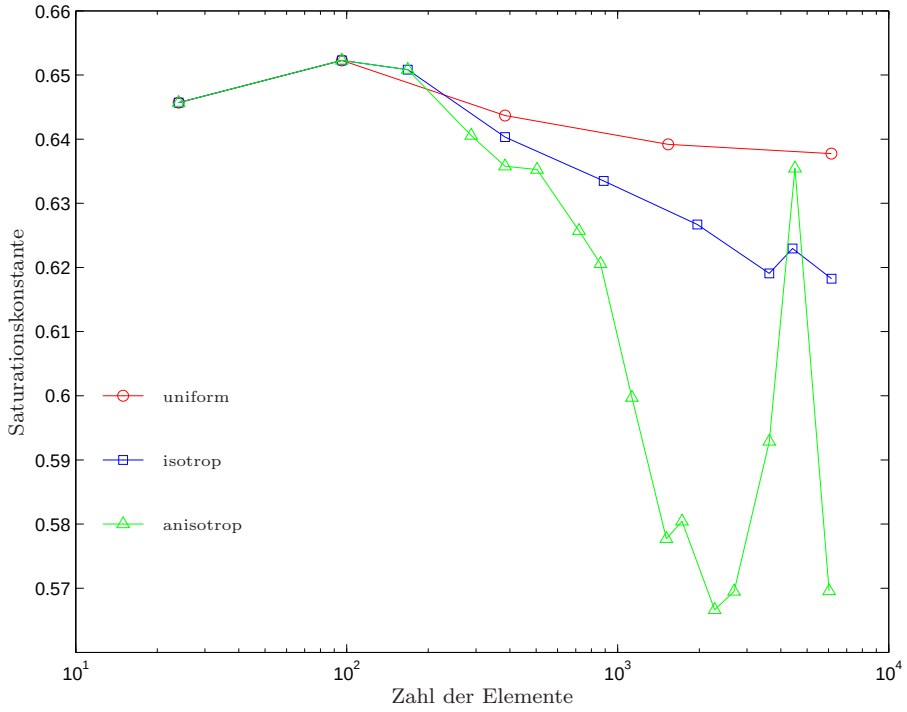


Abbildung 5.5: Experimentelle Saturationskonstante $q = |||\phi - \phi_{h/2}|||/|||\phi - \phi_h|||$ für das Kapazitätsproblem 5.2 für uniforme, isotrope und anisotrope Netzverfeinerung. In allen Fällen gilt $q < 1$, was die Zuverlässigkeit von η_H impliziert.

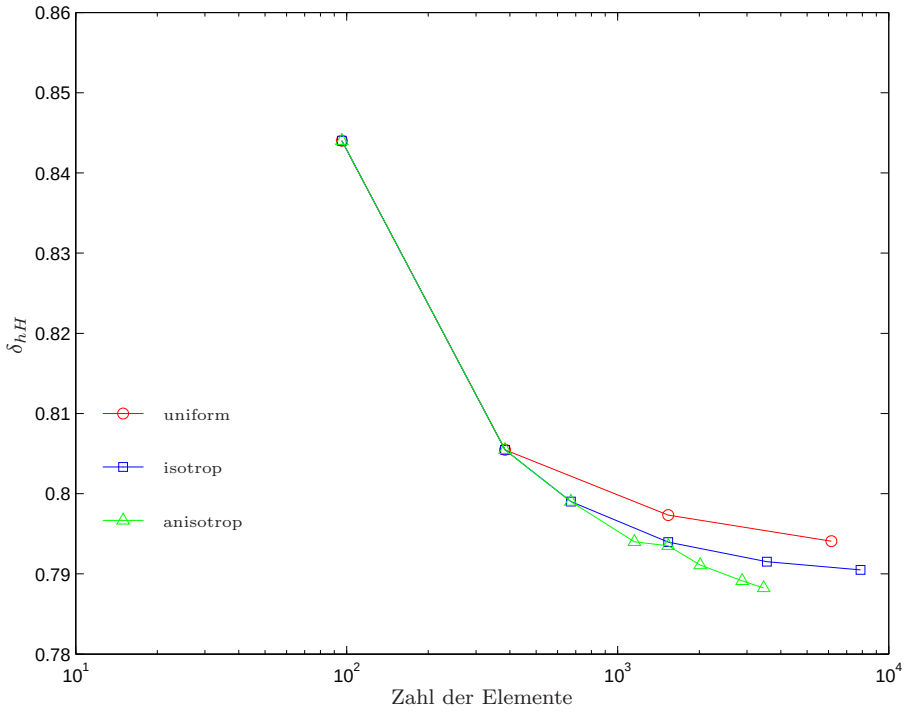


Abbildung 5.6: Experimentell ermittelte Verhältnisse $\delta_{hH} = |||\phi - \phi_h^{(1)}|||/|||\phi - \phi_h|||$ mit $\phi_h^{(1)} \in \mathcal{P}^1(\mathcal{T}_h)$ der Galerkinlösung für stückweise lineare Ansatzfunktionen in Kapazitätsproblem 5.2. Die Daten sind für eine uniforme sowie für $\widetilde{\mu}_M$-adaptive isotrope bzw. anisotrope Netzfolgen visualisiert.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
96	0.353	–	0.112	0.112	0.549	0.548
384	0.230	0.308	0.078	0.080	0.395	0.392
1536	0.148	0.319	0.052	0.053	0.263	0.260
6144	0.095	0.323	0.034	0.034	0.170	0.169

Tabelle 5.6: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine uniforme Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
96	0.3526	–	0.112	0.112	0.549	0.548
384	0.2300	0.308	0.078	0.080	0.395	0.392
672	0.1831	0.407	0.063	0.065	0.327	0.325
1536	0.1296	0.418	0.046	0.047	0.235	0.233
3552	0.0873	0.472	0.031	0.032	0.160	0.159
7872	0.0575	0.523	0.021	0.021	0.108	0.106
14496	0.0426	0.494	0.016	0.016	0.081	0.080
17664	0.0370	0.709	0.014	0.014	0.071	0.070

Tabelle 5.7: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_M$-adaptive isotrope Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

Lösung ein gewisser Fehler nicht vermieden werden kann. Die hohe Genauigkeit im uniformen Fall kann dadurch erklärt werden, dass die Werte ebendieser Netzfolge als Grundlage für die Extrapolation herangezogen wurden.

Eine weitere Größe, die von Interesse ist, ist das Verhältnis

$$\delta_{hH} = \frac{\|\|\phi - \phi_h^{(1)}\|\|}{\|\|\phi - \phi_{h/2}\|\|}$$

von dem die Effizienz und die Zuverlässigkeit von η_M abhängen. Hier bezeichnet $\phi_h^{(1)} \in \mathcal{P}^1(\mathcal{T}_h)$ die diskrete Lösung mit linearen Ansatzfunktionen. Für glatte Funktionen $\phi \in H^{1+\varepsilon}(\mathcal{T}_h)$ haben wir gezeigt, dass $\delta_{hH} = \mathcal{O}(h^\alpha)$ gilt. In Abbildung 5.6 sind die experimentell ermittelten Größen für uniforme und $\widetilde{\mu}_M$-adaptive isotrope bzw. anisotrope Verfeinerung über der Zahl der Rechtecke im feinen Gitter gezeichnet. Wir beobachten zwar eine Tendenz zur Verkleinerung von δ_{hH} mit wachsender Elementzahl und wachsender Anisotropie, jedoch ist eine Konvergenz gegen 0 nicht erkennbar. Die Beschränktheit von δ_{hH} impliziert die Effizienz des Fehlerschätzers η_M. Wie wir noch sehen werden, ist η_M zuverlässig und effizient. Diese Beobachtungen legen nahe, dass

$$\delta_{hH}^2 + q^2 < 1$$

gilt, was in Erath/Ferraz-Leite/Funken/Praetorius 2007 [12] für zweidimensionale Probleme experimentell bestätigt wird. Wir halten fest, dass Satz 3.11 die Äquivalenz von η_M und η_H als Schätzer für den Fehler auf dem groben Netz $\|\|\phi - \phi_h\|\|$ zeigt. In der Implementierung verwenden wir $\widetilde{\mu}_M$ als Indikator für $\mathcal{T}_h$, sodass mit der Zuverlässigkeit von η_H hier auch die Zuverlässigkeit von η_M zumindest für isotrope Netze folgen muss.

In Abbildung 5.7 sind der Fehler $\|\|\phi - \phi_h\|\|$ und die Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H und $\widetilde{\mu}_H$ für $\widetilde{\mu}_H$-adaptive isotrope Verfeinerungen über der Zahl der Elemente im groben Gitter $\mathcal{T}_h^{(\ell)}$ dargestellt. Als Referenz

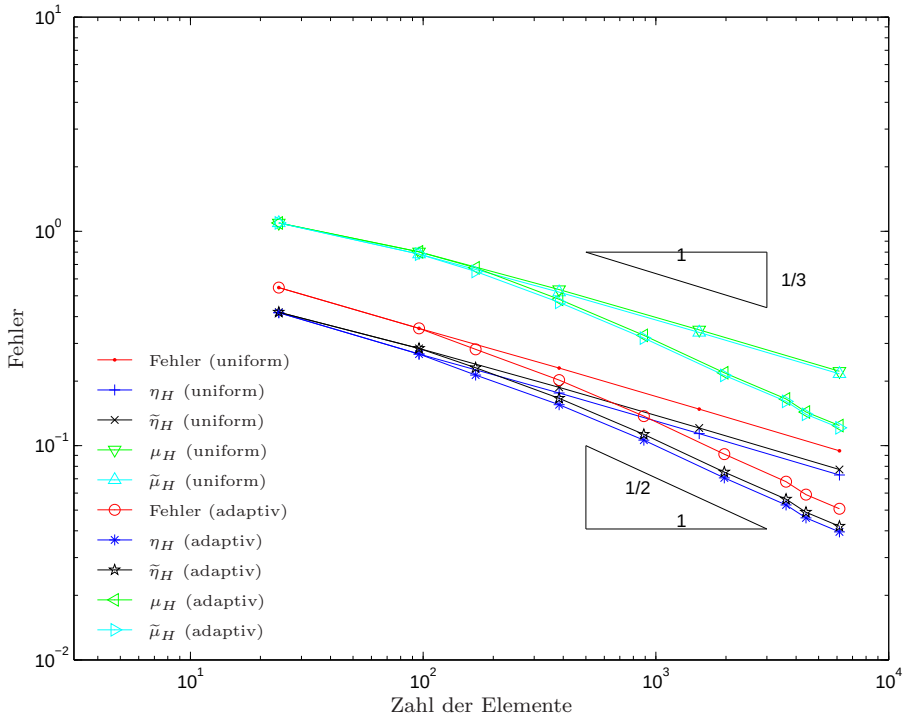


Abbildung 5.7: Fehler $\|\!|\phi - \phi_h|\!\|$ und Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \mu_H, \widetilde{\mu}_H$ für das Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope Netzverfeinerung mit fastregulären Paneelierungen.

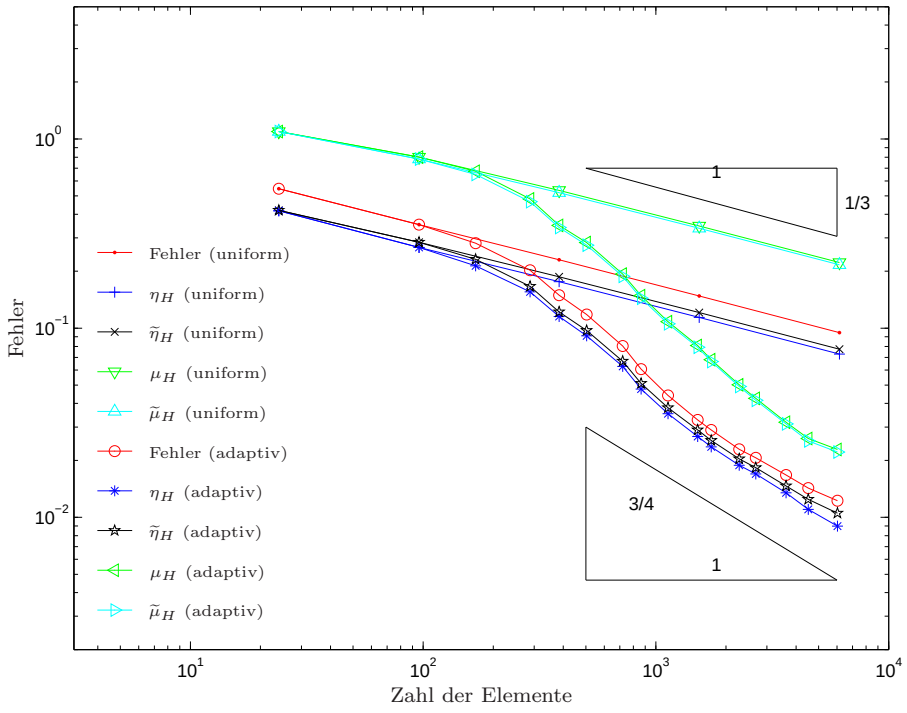


Abbildung 5.8: Fehler $\|\!|\phi - \phi_h|\!\|$ und Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \mu_H, \widetilde{\mu}_H$ im Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive anisotrope Verfeinerung mit fastregulären Paneelierungen.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
96	3.53×10^{-1}	–	1.11×10^{-1}	1.12×10^{-1}	5.49×10^{-1}	5.48×10^{-1}
384	2.30×10^{-1}	0.308	7.82×10^{-2}	7.96×10^{-2}	3.95×1^{-1}	3.92×10^{-1}
672	1.83×10^{-1}	0.407	6.37×10^{-2}	6.47×10^{-2}	3.27×10^{-1}	3.25×10^{-1}
1152	1.30×10^{-1}	0.640	4.60×10^{-2}	4.68×10^{-2}	2.35×10^{-1}	2.33×10^{-1}
1536	9.49×10^{-2}	1.087	3.39×10^{-2}	3.46×10^{-2}	1.71×10^{-1}	1.70×10^{-1}
2016	7.49×10^{-2}	0.868	2.72×10^{-2}	2.77×10^{-2}	1.39×10^{-1}	1.37×10^{-1}
2880	5.04×10^{-2}	1.113	1.88×10^{-2}	1.92×10^{-2}	9.49×10^{-2}	9.39×10^{-2}
3456	3.77×10^{-2}	1.588	1.45×10^{-2}	1.49×10^{-2}	7.29×10^{-2}	7.20×10^{-2}
4512	2.64×10^{-2}	1.331	1.09×10^{-2}	1.13×10^{-2}	5.37×10^{-2}	5.27×10^{-2}
6048	1.89×10^{-2}	1.144	8.60×10^{-3}	9.20×10^{-3}	4.09×10^{-2}	3.96×10^{-2}
6912	1.68×10^{-2}	0.881	7.52×10^{-3}	8.09×10^{-3}	3.58×10^{-2}	3.33×10^{-2}
9120	1.30×10^{-2}	0.938	6.08×10^{-3}	-	-	2.46×10^{-2}
10752	1.18×10^{-2}	0.596	5.48×10^{-3}	-	-	2.08×10^{-2}

Tabelle 5.8: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_M$-adaptive anisotrope Netzfolge in Kapazitätsproblem 5.2. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

sind die Daten auch für eine uniforme Netzfolge abgebildet. Die genauen Zahlen lassen sich in den Tabellen 5.3 und 5.4 einsehen, wo auch die experimentelle Konvergenzrate α eingetragen ist. Wir definieren den im ℓ-ten Schritt gemessenen Fehler $e_\ell := \|\|\phi - \phi_h^{(\ell)}\|\|$. Mit $N_\ell = \mathrm{card}\mathcal{T}_h^{(\ell)}$ bezeichnen wir die Zahl der Randelemente des Netzes im ℓ-ten Schritt. Mit dem Ansatz $e_j = CN_j^{-\alpha}$ für $j \in \{\ell-1, \ell\}$ lässt sich die experimentelle Konvergenzrate durch

$$\alpha = -\frac{\log(e_{\ell-1}/e_\ell)}{\log(N_{\ell-1}/N_\ell)}$$

berechnen. Wir beobachten für eine uniforme Netzfolge, dass der Verfahrensfehler mit etwa der Ordnung 1/3 konvergiert. Mit Hilfe der adaptiven isotropen Verfeinerungsstrategie kann die Konvergenz auf etwa 1/2 verbessert werden. Der Gewinn ist spürbar, jedoch noch weit von der optimalen Konvergenzordnung $\alpha = 3/4$ entfernt. Es scheinen Kantensingularitäten vorzuliegen, die durch isotrope Netzfolgen nicht hinreichend gut approximiert werden können. Das Ergebnis desselben Experimentes mit anisotroper Verfeinerung ist in Abbildung 5.8 zu sehen. In Tabelle 5.5 sind die genauen Werte der Fehlerschätzer sowie die experimentelle Konvergenzordnung angeführt. Wir erkennen am Graphen, dass die anisotrope Verfeinerungsstrategie tendenziell zu optimaler Konvergenz führt. Da das Verhalten aber nicht mehr so regelmäßig wie im uniformen oder isotropen Fall ist, kann dies nur schwer an den Zahlen abgelesen werden. Die Zuverlässigkeit und Effizienz aller Fehlerschätzer lässt sich empirisch verifizieren, da alle Kurven parallel zum Fehler verlaufen.

In Abbildung 5.9 sind der Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und die Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M und $\widetilde{\mu}_M$ für $\widetilde{\mu}_M$-adaptive isotrope Verfeinerungen über der Zahl der Rechtecke im feinen Gitter $\mathcal{T}_{h/2}^{(\ell)}$ zu sehen. Die Daten, die aus einer uniformen Netzfolge hervorgehen, sind ebenfalls abgebildet. Die genauen Zahlen sind in den Tabellen 5.6 und 5.7 festgehalten, wo auch die experimentelle Konvergenzrate angeführt ist. So wie bei der $\widetilde{\mu}_H$-adaptiven Strategie verbessert sich die Konvergenzrate auch mit $\widetilde{\mu}_M$ als Indikator von etwa $\alpha = 1/3$ im uniformen Fall auf etwa $\alpha = 1/2$ im adaptiven isotropen Fall. In Abbildung 5.10 und Tabelle 5.8 sind die Daten für die $\widetilde{\mu}_M$-adaptive anisotrope Strategie abgebildet bzw. angeführt. Wie bereits in Kapitel 4 erklärt, ist die Berechnung der Systemmatrix für $\mathcal{P}^1$-Elemente problematisch, da Rundungsfehler, die bei der analytischen Berechnung der Integrale auftreten, signifikant werden können. Daher sind die Daten nur so weit, wie sie als zuverlässig erachtet werden können bzw. berechenbar sind, dargestellt. Da zur Ermittlung von $\widetilde{\mu}_H$ die genannte Systemmatrix nicht benötigt wird,

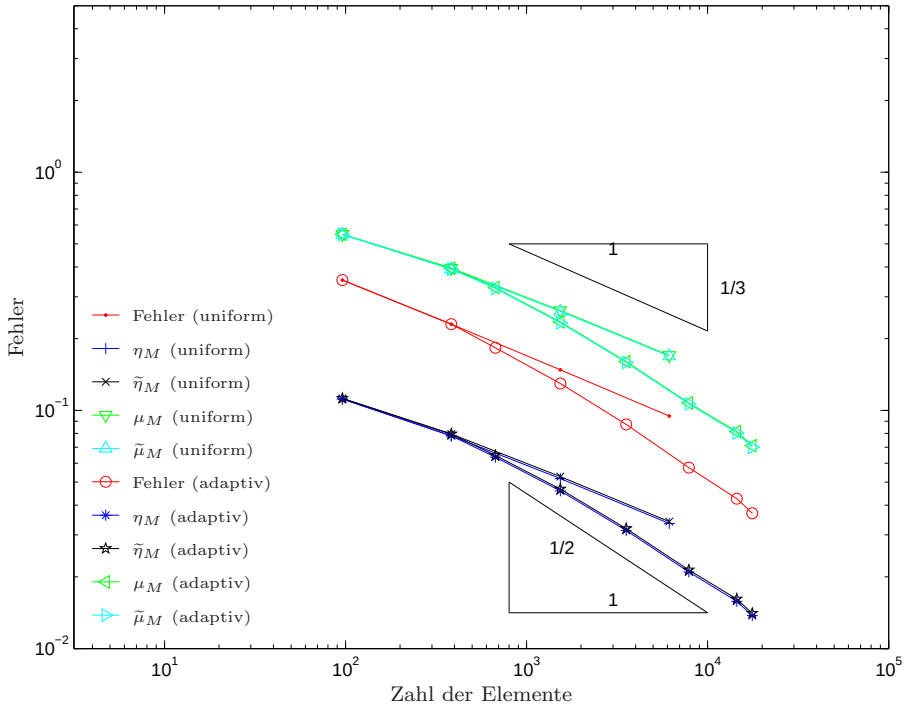


Abbildung 5.9: Fehler $\|\!|\phi - \phi_{h/2}|\!\|$ und Fehlerschätzer $\eta_M, \widetilde{\eta}_M, \mu_M$ sowie $\widetilde{\mu}_M$ im Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_M$-adaptive isotrope Verfeinerungen.

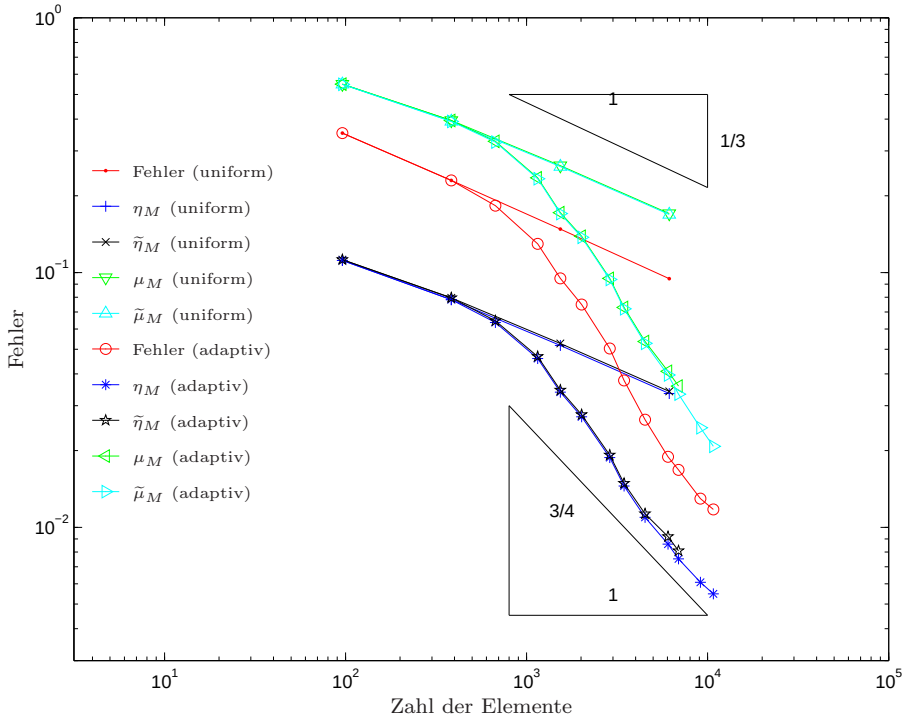


Abbildung 5.10: Fehler $\|\!|\phi - \phi_{h/2}|\!\|$ und Fehlerschätzer $\eta_M, \widetilde{\eta}_M, \mu_M, \widetilde{\mu}_M$ im Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_M$-adaptive anisotrope Verfeinerung.

ist das Verhalten dieses Schätzers am stabilsten. Wieder erkennen wir analog zur Verwendung von $\widetilde{\mu}_H$ als Indikator eine Verbesserung der Konvergenzordnung auf etwa $\alpha = 3/4$, was dem Optimum entspricht. Die Zuverlässigkeit und Effizienz aller Fehlerschätzer kann empirisch bestätigt werden, da alle Kurven parallel zum Fehler verlaufen.

Wir haben gesehen, dass aus η_H bzw. η_M durch Verwendung der L^2-Projektion die äquivalenten, aber leichter zu berechnenden Schätzer $\widetilde{\eta}_H$ bzw. $\widetilde{\eta}_M$ gewonnen werden können. In Satz 3.7 haben wir

$$\eta_H \leq \widetilde{\eta}_H \quad \text{und} \quad \eta_M \leq \widetilde{\eta}_M$$

bewiesen. Analog haben wir auch die dualen Zusammenhänge

$$\widetilde{\mu}_H \leq \mu_H \quad \text{und} \quad \widetilde{\mu}_M \leq \mu_M$$

für die lokalen Fehlerschätzer in Satz 3.10 gezeigt. Es stellt sich die Frage, inwiefern die Verwendung der Galerkin- bzw. L^2-Projektion Einfluss auf die Größenordnung der Schätzer hat. Tabelle 5.9 führt die Quotienten

$$\widetilde{\eta}_H/\eta_H \quad \text{und} \quad \mu_H/\widetilde{\mu}_H$$

an. Im uniformen und im isotropen Fall unterscheidet sich $\widetilde{\eta}_H$ im Verhältnis nur um etwa 0.06 von η_H. Für die anisotrope Netzverfeinerung beobachten wir bei zunehmender Elementzahl eine Vergrößerung des Quotienten auf etwa 0.17. Interessanterweise ist eine solche Verstärkung der Abweichung für die Schätzer μ_H und $\widetilde{\mu}_H$ nicht zu beobachten, wo in allen Fällen das Verhältnis etwa 0.03 beträgt. In Tabelle 5.10 sind die Quotienten

$$\widetilde{\eta}_M/\eta_M \quad \text{und} \quad \mu_M/\widetilde{\mu}_M$$

angeführt. Wir erkennen auch hier, dass die Unterschiede im uniformen und isotropen Fall marginal bleiben. Der Unterschied zwischen η_M und $\widetilde{\eta}_M$ beträgt im Verhältnis nur etwa 0.02. Im anisotropen Fall verstärkt sich jedoch der Einfluß der L^2-Projektion, sodass der Quotient beinahe 0.08 beträgt. Bei μ_M und $\widetilde{\mu}_M$ verhält es sich ähnlich. Hier nimmt die Abweichung von etwa $0.01 - 0.02$ im uniformen bzw. isotropen Fall auf beinahe 0.08 zu. Dabei beobachten wir einen Sprung im letzten Schritt, bei dem die Rundungsfehler in der $\mathcal{P}^1$-Systemmatrix scheinbar irrelevant sind. Möglicherweise ist dieser Sprung aber ein Ausdruck der sich ankündigenden Instabilität, die in Folge bei noch feiner werdendem Netz beobachtet wird.

In Abbildung 5.11 ist ein Ausschnitt der Netzfolge zu sehen, die durch $\widetilde{\mu}_H$-Adaptivität entsteht. Im isotropen Fall, im Bild links, ist zu erkennen, dass die Verfeinerungen primär zu den Ecken hin verlaufen. Im anisotropen Fall, im Bild rechts, erkennen wir starke Verfeinerungen zu den Kanten hin. Das Zulassen anisotroper Elemente ermöglicht das effektive Approximieren von Kantensingularitäten. In Abbildung 5.12 ist ein Ausschnitt der durch $\widetilde{\mu}_M$-Adaptivität entstandenen Netzfolge zu sehen. Wir beobachten hier dasselbe Verhalten.

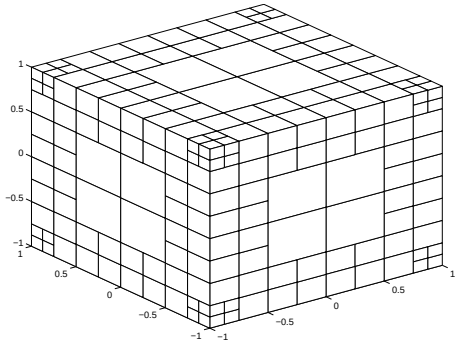
Abschließend kann man festhalten, dass wir in diesem Beispiel beobachten konnten, dass die Fehlerschätzer wie von der Theorie vorhergesagt im isotropen Fall zuverlässig und effizient sind. Darüberhinaus erhält man im anisotropen Fall optimale Konvergenz und die Formregularität des Netzes $\sigma(\mathcal{T}_h)$ scheint keinen Einfluss auf die Güte der Schätzer zu haben.

uniform			isotrop			anisotrop		
N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$	N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$	N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$
24	1.005	1.000	24	1.005	1.000	24	1.005	1.000
96	1.063	1.024	96	1.063	1.024	96	1.064	1.024
384	1.062	1.027	168	1.081	1.040	168	1.081	1.040
1536	1.061	1.028	384	1.071	1.034	288	1.070	1.033
6144	1.062	1.028	888	1.065	1.030	384	1.058	1.025
			1968	1.063	1.028	504	1.067	1.032
			3624	1.064	1.027	720	1.065	1.029
			4416	1.062	1.025	864	1.070	1.030
			6150	1.063	1.024	1128	1.075	1.026
						1512	1.083	1.020
						1728	1.079	1.022
						2280	1.080	1.021
						2688	1.077	1.024
						3624	1.086	1.021
						4512	1.170	1.030

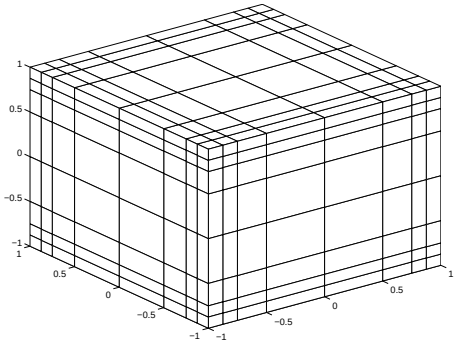
Tabelle 5.9: Verhältnisse $\widetilde{\eta}_H/\eta_H$ und $\mu_H/\widetilde{\mu}_H$ in Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzverfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

uniform			isotrop			anisotrop		
N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$	N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$	N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$
96	1.006	1.001	96	1.006	1.001	96	1.006	1.001
384	1.017	1.008	384	1.017	1.008	384	1.017	1.008
2536	1.018	1.009	672	1.016	1.007	672	1.016	1.007
6144	1.019	1.009	1536	1.017	1.008	1152	1.017	1.008
			3552	1.018	1.030	1536	1.019	1.009
			7872	1.020	1.028	2016	1.019	1.009
			14496	1.023	1.027	2880	1.021	1.011
			17664	1.023	1.025	3456	1.024	1.012
						4512	1.033	1.017
						6048	1.070	1.033
						6912	1.075	1.075

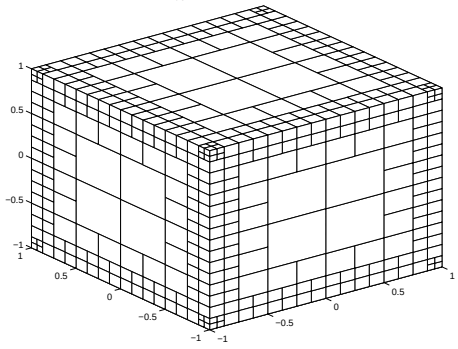
Tabelle 5.10: Verhältnisse $\widetilde{\eta}_H/\eta_H$ und $\mu_H/\widetilde{\mu}_H$ in Kapazitätsproblem 5.2 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzverfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.



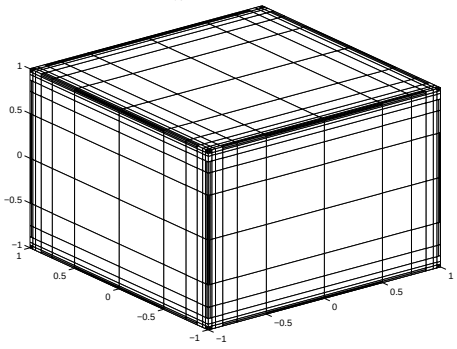
$\mathcal{T}_h^{(3)}, N = 384$



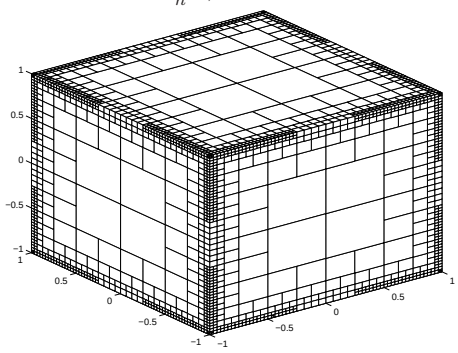
$\mathcal{T}_h^{(4)}, N = 384$



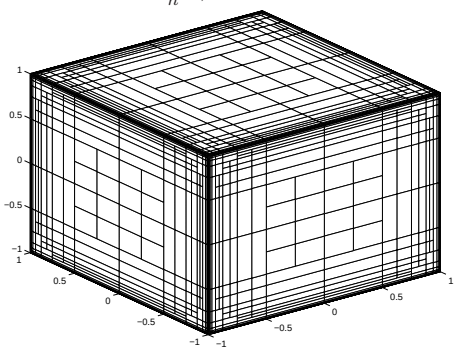
$\mathcal{T}_h^{(4)}, N = 888$



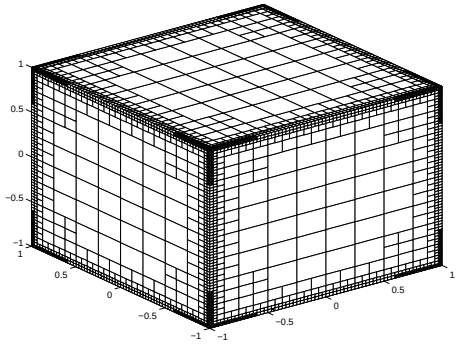
$\mathcal{T}_h^{(8)}, N = 1128$



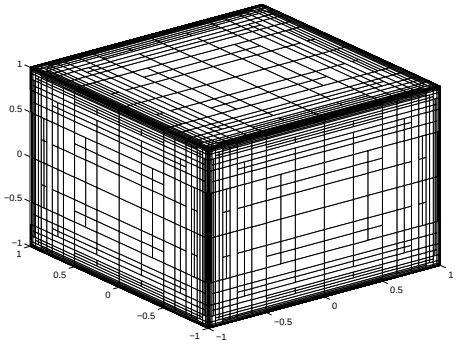
$\mathcal{T}_h^{(6)}, N = 3624$



$\mathcal{T}_h^{(12)}, N = 2688$

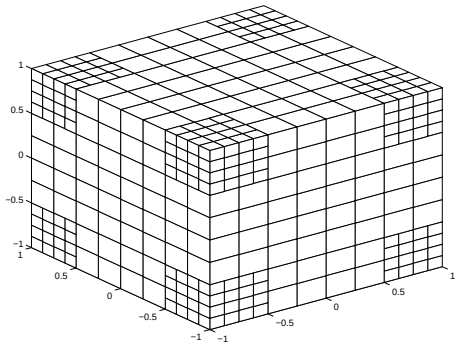


$\mathcal{T}_h^{(8)}, N = 6150$

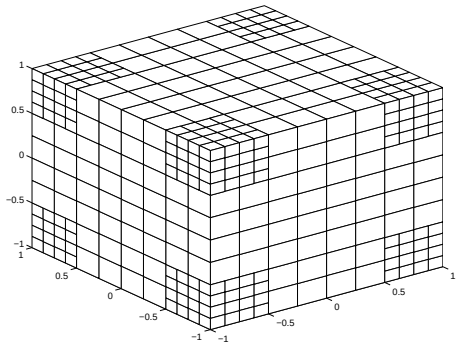


$\mathcal{T}_h^{(14)}, N = 4512$

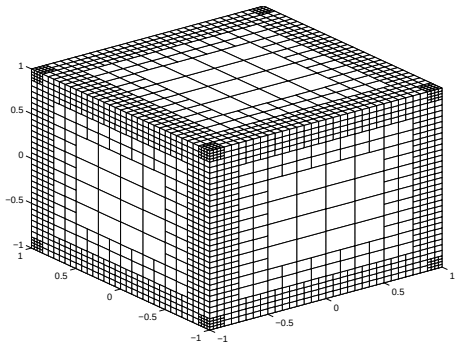
Abbildung 5.11: Ausschnitt der Folge von Netzen, die durch $\widetilde{\mu}_H$-adaptive isotrope (links) bzw. anisotrope (rechts) Verfeinerungen im Kapazitätsproblem 5.2 hervorgeht.



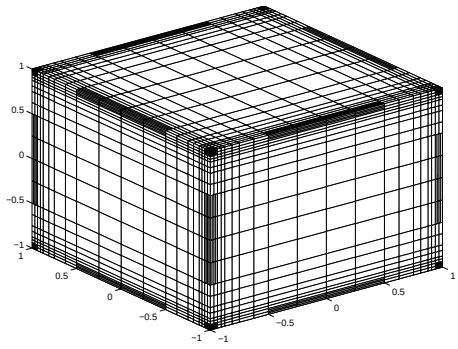
$\mathcal{T}_{h/2}^{(2)}, N = 672$



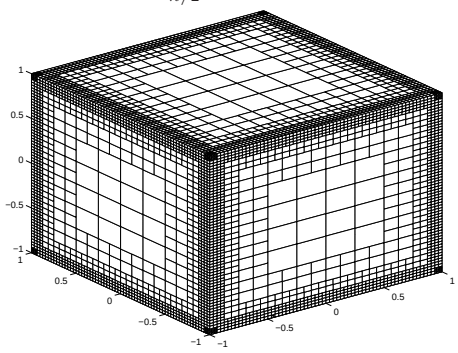
$\mathcal{T}_{h/2}^{(2)}, N = 672$



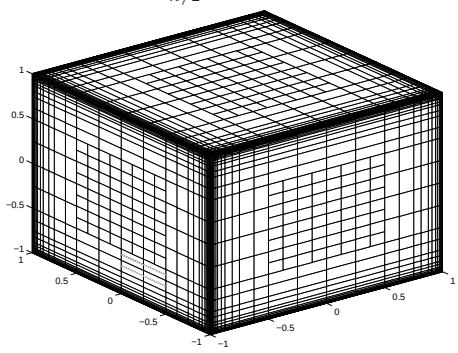
$\mathcal{T}_{h/2}^{(4)}, N = 3552$



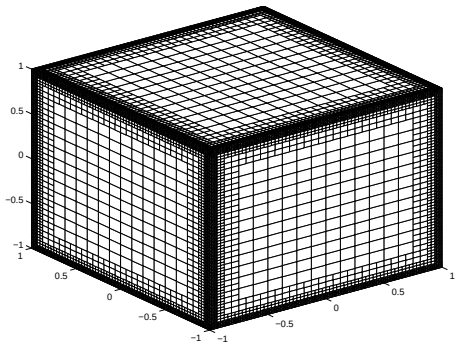
$\mathcal{T}_{h/2}^{(6)}, N = 2880$



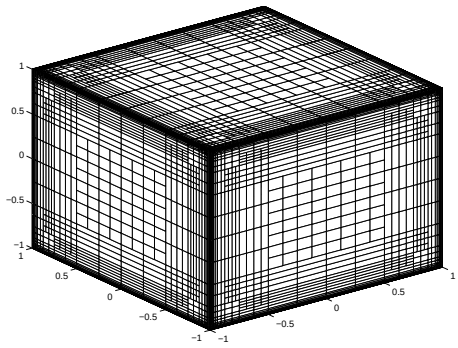
$\mathcal{T}_{h/2}^{(5)}, N = 7872$



$\mathcal{T}_{h/2}^{(9)}, N = 6048$



$\mathcal{T}_{h/2}^{(7)}, N = 17664$



$\mathcal{T}_{h/2}^{(12)}, N = 10752$

Abbildung 5.12: Ausschnitt der Folge von Netzen, die durch $\widetilde{\mu}_M$-adaptive isotrope (links) bzw. anisotrope (rechts) Verfeinerungen im Kapazitätsproblem 5.2 hervorgeht.

uniform		isotrop		anisotrop	
N	q	N	q	N	q
12	0.7383	12	0.7383	12	0.7383
48	0.7299	27	0.7323	27	0.7323
192	0.7214	54	0.7298	46	0.7296
768	0.7161	129	0.7187	75	0.7183
3072	0.7127	240	0.7149	94	0.7163
		315	0.7117	138	0.7062
		618	0.7074	173	0.7094
		774	0.7047	224	0.6980
		1353	0.6983	296	0.6839
		1743	0.6958	344	0.6688
		2913	0.6904	428	0.6373
		3870	0.6881	532	0.6196
				660	0.5877
				811	0.5450
				992	0.5144
				1205	0.4721
				1442	0.4470
				1860	0.4086
				2251	0.3680
				2846	0.3111

Tabelle 5.11: Experimentelle Saturationskonstante $q = \frac{|||\phi-\phi_{h/2}|||}{|||\phi-\phi_h|||}$ in Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Verfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

5.3 Schirmproblem auf L-förmigem Randstück

Unter einem Schirm versteht man ein ebenes Flächenstück im dreidimensionalen Raum. Wir wollen die Symmsche Integralgleichung auf dem L-förmigen Flächenstück $\Gamma = (-1,1)^2 \setminus (0,1)^2 \times \{0\}$ wiederum mit konstanter rechter Seite $f = 1$ lösen. Extrapolation mit der Aitkinschen Δ^2-Methode führt zu einer geschätzten Energienorm von $|||\phi|||^2 = 8.284570593849$. Das grobe Startgitter $\mathcal{T}_h^{(0)}$ mit $N = 12$ Rechtecken ist in Abbildung 5.13 zu sehen. In Abbildung 5.14 ist exemplarisch eine diskrete Lösung $\phi_h^{(19)}$ über dem zugehörigen Netz $\mathcal{T}_h^{(19)}$ abgebildet. Diese Lösung wurde mit der $\widetilde{\mu}_H$-adaptiven anisotropen Strategie ermittelt. Wir erkennen starke Singularitäten entlang aller Kanten und an den konvexen Ecken.

Wie bereits in Kapazitätsproblem 5.2 wollen wir auch für das Schirmproblem die Saturationsannahme empirisch überprüfen. In Tabelle 5.11 sind die experimentellen Größen

$$q = \frac{|||\phi - \phi_{h/2}|||}{|||\phi - \phi_h|||}$$

für eine uniforme und für aus $\widetilde{\mu}_H$-adaptiver isotroper und anisotroper Strategie hervorgegangene Netzfolgen angeführt. In Abbildung 5.15 sind die Werte als Graphen über der Zahl der Elemente im groben Netz dargestellt. Offensichtlich sind die experimentellen Konstanten nach oben durch 0.75 beschränkt, was empirisch die Saturationsannahme und somit die Zuverlässigkeit von η_H zeigt.

uniform			isotrop			anisotrop		
N	$f(q)$	C_{rel}	N	$f(q)$	C_{rel}	N	$f(q)$	C_{rel}
12	1.482	1.482	12	1.482	1.482	12	1.482	1.482
48	1.463	1.463	27	1.469	1.469	27	1.469	1.469
192	1.444	1.444	54	1.463	1.463	46	1.462	1.462
768	1.433	1.433	129	1.438	1.438	75	1.437	1.438
3072	1.426	1.426	240	1.430	1.431	94	1.433	1.434
			315	1.424	1.425	138	1.412	1.414
			618	1.415	1.416	173	1.419	1.422
			774	1.409	1.412	224	1.397	1.401
			1353	1.397	1.404	296	1.371	1.376
			1743	1.392	1.400	344	1.345	1.351
			2913	1.382	1.393	428	1.298	1.304
			3870	1.378	1.389	532	1.274	1.281
						660	1.236	1.245
						811	1.193	1.200
						992	1.166	1.174
						1205	1.134	1.142
						1442	1.118	1.127
						1860	1.096	1.105
						2251	1.076	1.087
						2846	1.052	1.064

Tabelle 5.12: Experimentelle Zuverlässigkeitskonstante C_{rel} und $f(q) = 1/\sqrt{1-q^2}$ für die experimentelle Saturationskonstante q in Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzfolgen. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

N	$\|\!\|\!\|\phi - \phi_h\|\!\|\!\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
12	0.665	–	0.448	0.461	1.420	1.410
48	0.491	0.219	0.335	0.347	1.088	1.075
192	0.358	0.227	0.248	0.256	0.801	0.790
768	0.258	0.236	0.180	0.186	0.580	0.572
3072	0.185	0.241	0.130	0.134	0.416	0.410

Tabelle 5.13: Fehler $|\!|\!|\phi - \phi_h|\!|\!|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine uniforme Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

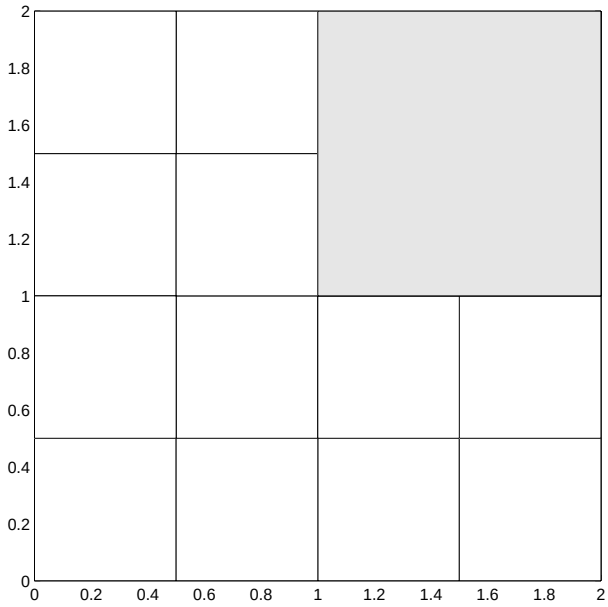


Abbildung 5.13: Startnetz $\mathcal{T}_h^{(0)}$ mit $N = 12$ Elementen in Beispiel 5.3.

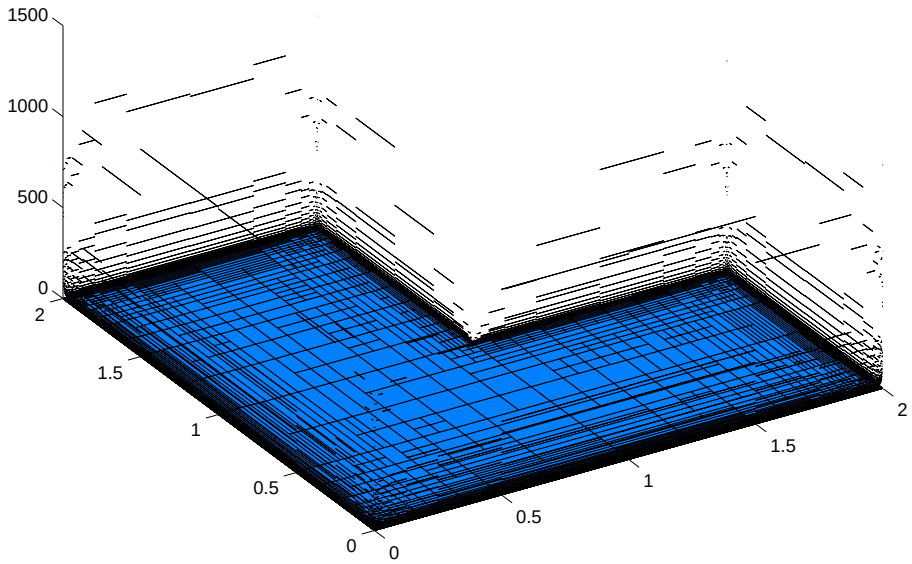


Abbildung 5.14: Diskrete Lösung $\phi_h^{(19)}$ über der Triangulierung $\mathcal{T}_h^{(19)}$ mit $N = 2846$ Elementen, die durch Algorithmus 5.1 in Beispiel 5.3 mit $\widetilde{\mu}_H$ als Indikator und anisotroper Verfeinerung generiert wurde. Die Lösung weist starke Singularitäten entlang der Kanten und an den konvexen Ecken auf.

N	$\|\|\|\phi - \phi_h\|\|\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
12	0.665	–	0.448	0.461	1.420	1.410
27	0.540	0.256	0.368	0.383	1.202	1.183
54	0.444	0.284	0.303	0.319	1.013	0.989
129	0.341	0.301	0.237	0.246	0.773	0.760
240	0.278	0.329	0.194	0.201	0.632	0.622
315	0.248	0.419	0.174	0.180	0.563	0.554
618	0.188	0.409	0.133	0.137	0.429	0.423
774	0.172	0.409	0.122	0.125	0.392	0.387
1353	0.134	0.447	0.095	0.098	0.308	0.304
1743	0.121	0.417	0.086	0.089	0.278	0.274
2913	0.095	0.468	0.068	0.070	0.220	0.217
3870	0.084	0.416	0.061	0.062	0.196	0.194

Tabelle 5.14: Fehler $\|\|\|\phi-\phi_h\|\|\|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_H$-adaptive isotrope Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

In Tabelle 5.12 werden die experimentelle Zuverlässigkeitskonstante

$$C_{\mathrm{rel}} = \frac{|||\phi - \phi_h|||}{\eta_H}$$

und die Funktion

$$f(q) = \frac{1}{\sqrt{1-q^2}}$$

der experimentellen Saturationskonstante direkt miteinander verglichen. Die Daten stammen aus Experimenten mit uniformer bzw. $\widetilde{\mu}_H$-adaptiver Netzverfeinerungsstrategie. Analytisch sollte $f(q) = C_{\mathrm{rel}}$ für die experimentellen Größen gelten. Wie bereits im Kapazitätsproblem 5.2 beobachten wir auch für den L-förmigen Schirm leichte Abweichungen, die vermutlich auf die ungenaue Extrapolation der Energienorm $|||\phi|||$ zurückzuführen ist. Da die uniforme Netzfolge als Grundlage für die Extrapolation herangezogen wurde, ist klar, dass für diesen Fall die Daten exakter sind als für den isotropen bzw. den anisotropen Fall.

In Abbildung 5.16 sind der Fehler $|||\phi-\phi_h|||$ und die Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \mu_H$ und $\widetilde{\mu}_H$ für $\widetilde{\mu}_H$-adaptive isotrope Verfeinerungen über der Zahl der Rechtecke in $\mathcal{T}_h^{(\ell)}$ dargestellt. Zur Illustration sind ebenfalls die Daten für eine uniforme Netzfolge abgebildet. Die genauen Zahlen können den Tabellen 5.13 und 5.14 entnommen werden, in denen auch die experimentelle Konvergenzrate α des Fehlers eingetragen ist. Wir erkennen eine Verbesserung der Konvergenzrate von etwa $\alpha = 1/4$ im uniformen Fall auf nahezu $\alpha = 1/2$ im isotropen Fall. Wie zu erwarten ist, kann mit einer isotropen Strategie die optimale Konvergenzrate von $\alpha = 3/4$ nicht erreicht werden, da die Kantensingularitäten nicht effektiv approximiert werden. Das Ergebnis einer Simulation mit der $\widetilde{\mu}_H$-adaptiven anisotropen Strategie ist in Abbildung 5.17 zu sehen. Die zugehörigen Zahlen können Tabelle 5.15 entnommen werden. Wir erkennen, dass nach einer anfänglichen vorasymptotischen Phase vor allem in den letzten gerechneten Schritten die optimale Konvergenz deutlich erkennbar ist. Der Gewinn gegenüber dem uniformen Fall ist offensichtlich. Die Zuverlässigkeit und Effizienz aller Fehlerschätzer lässt sich empirisch anhand der Graphen verifizieren, alle Kurven verlaufen parallel zum Fehler.

In Abbildung 5.18 sind der Fehler $|||\phi - \phi_{h/2}|||$ und die Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M sowie $\widetilde{\mu}_M$ über der Zahl der Rechtecke im feinen Netz $\mathcal{T}_{h/2}$ für eine uniforme und eine $\widetilde{\mu}_M$-adaptive isotrope Folge gezeichnet. In Abbildung 5.19 sind dieselben Daten für die aus der $\widetilde{\mu}_M$-adaptiven anisotropen Strategie

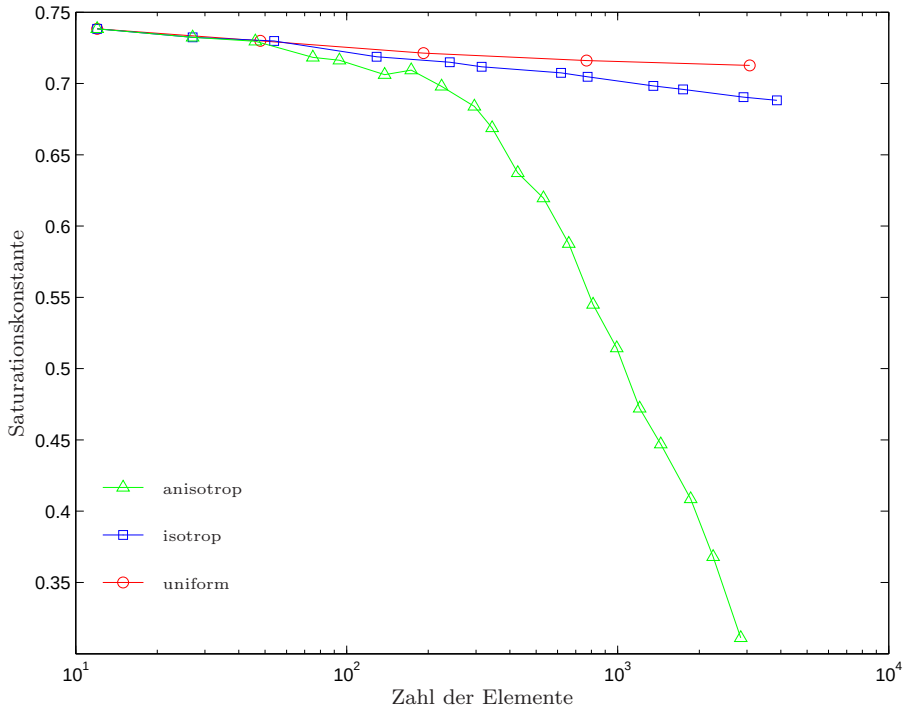


Abbildung 5.15: Experimentelle Saturationskonstante $q = |||\phi - \phi_{h/2}|||/|||\phi - \phi_h|||$ für das Beispiel 5.3 für uniforme, isotrope und anisotrope Netzverfeinerung. In allen Fällen gilt $q < 1$, was die Zuverlässigkeit von η_H impliziert.

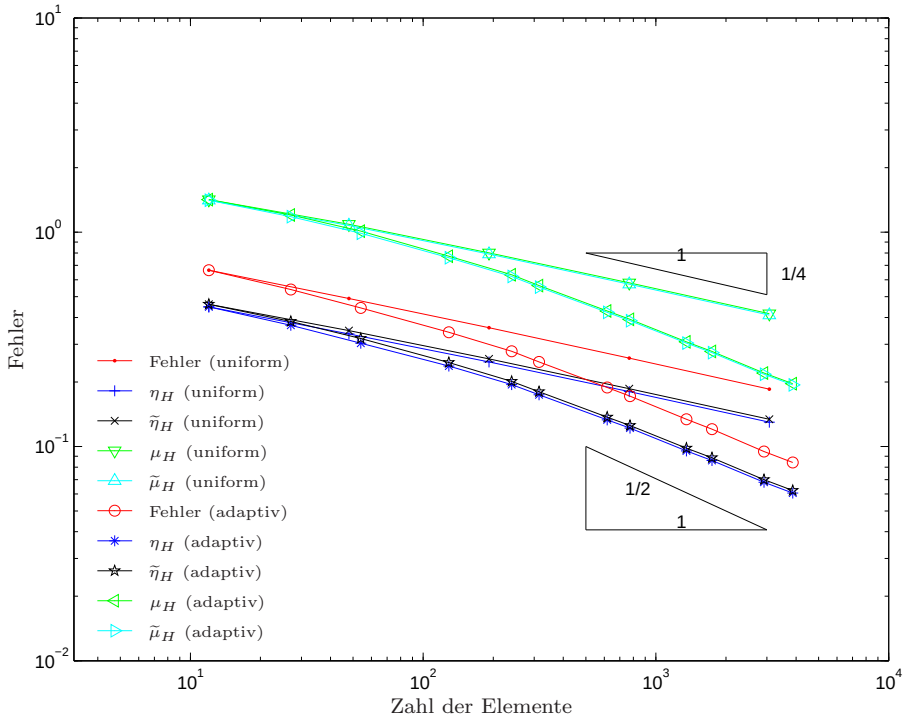


Abbildung 5.16: Fehler $|||\phi - \phi_h|||$ und Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \mu_H, \widetilde{\mu}_H$ im Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope Verfeinerungen.

N	$\|\|\|\phi - \phi_h\|\|\|$	α	η_H	$\widetilde{\eta}_H$	μ_H	$\widetilde{\mu}_H$
12	0.665	–	0.448	0.461	1.420	1.410
27	0.540	0.256	0.368	0.383	1.202	1.183
46	0.444	0.370	0.303	0.319	1.012	0.989
75	0.350	0.488	0.243	0.253	0.792	0.778
94	0.289	0.845	0.202	0.209	0.660	0.648
138	0.219	0.716	0.155	0.161	0.496	0.488
173	0.176	0.979	0.124	0.128	0.405	0.398
224	0.134	1.045	0.096	0.099	0.308	0.303
296	0.101	1.040	0.073	0.076	0.232	0.229
344	0.080	1.536	0.059	0.062	0.188	0.185
428	0.062	1.123	0.048	0.050	0.150	0.148
532	0.049	1.155	0.038	0.040	0.117	0.115
660	0.039	1.030	0.032	0.033	0.092	0.091
811	0.032	0.988	0.027	0.028	0.073	0.072
992	0.026	0.946	0.023	0.024	0.059	0.058
1205	0.022	0.841	0.020	0.021	0.047	0.047
1442	0.019	0.824	0.017	0.018	0.039	0.039
1860	0.016	0.667	0.015	0.016	0.031	0.031
2251	0.014	0.760	0.013	0.014	0.026	0.025
2846	0.012	0.735	0.011	0.013	0.021	0.021

Tabelle 5.15: Fehler $\|\|\|\phi-\phi_h\|\|\|$ und Fehlerschätzer η_H, $\widetilde{\eta}_H$, μ_H, $\widetilde{\mu}_H$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_H$-adaptive anisotrope Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
48	0.491	–	0.137	0.139	0.712	0.706
192	0.358	0.227	0.104	0.106	0.544	0.538
768	0.258	0.236	0.076	0.078	0.400	0.395
3072	0.185	0.241	0.055	0.057	0.290	0.286
12288	0.132	0.244	0.040	0.041	0.208	0.205

Tabelle 5.16: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine uniforme Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

gewonnenen Netzfolge abgebildet. Die genauen Zahlen lassen sich den Tabellen 5.16, 5.17 und 5.18 entnehmen, wo auch die experimentelle Konvergenzrate α des Fehlers eingetragen ist. Analog zum $\widetilde{\mu}_H$-adaptiven Fall erkennen wir auch bei $\widetilde{\mu}_M$-adaptiver Verfeinerung eine Verbesserung der Konvergenz. Im isotropen Fall wird beinahe eine Rate von $\alpha = 1/2$ erreicht. Im anisotropen Fall scheint das Experiment nicht hinreichend weit durchführbar zu sein, um aus einer hyperkonvergenten präasymptotischen Phase hinauszukommen. Wir beobachten allerdings am Graphen, dass sich die Konvergenz scheinbar in Richtung $\alpha = 3/4$ einpendelt, was dem optimalen asymptotischen Verhalten entspricht. Wir halten fest, dass alle Kurven parallel zu der des Fehlers verlaufen, daher sind die Fehlerschätzer empirisch zuverlässig und effizient.

Wie bereits in Kapazitätsproblem 5.2 wollen wir uns auch für das vorliegende Schirmproblem der Frage widmen, inwiefern der Austausch von L^2- bzw. Galerkinprojektion Einfluss auf die Schätzer

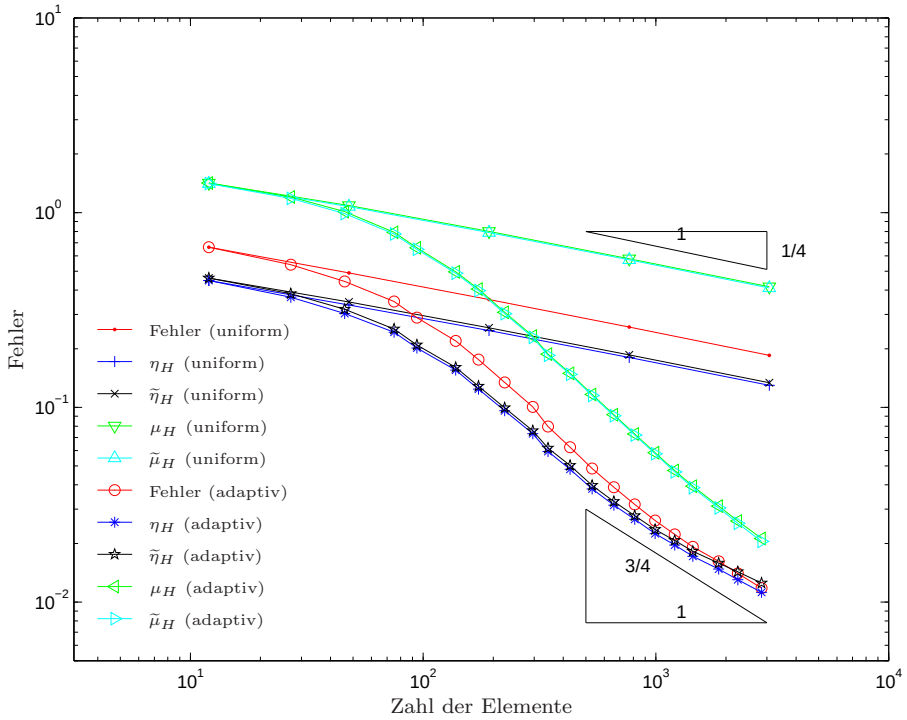


Abbildung 5.17: Fehler $|||\phi - \phi_h|||$ und Fehlerschätzer $\eta_H, \widetilde{\eta}_H, \mu_H, \widetilde{\mu}_H$ in Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive anisotrope Netzverfeinerung.

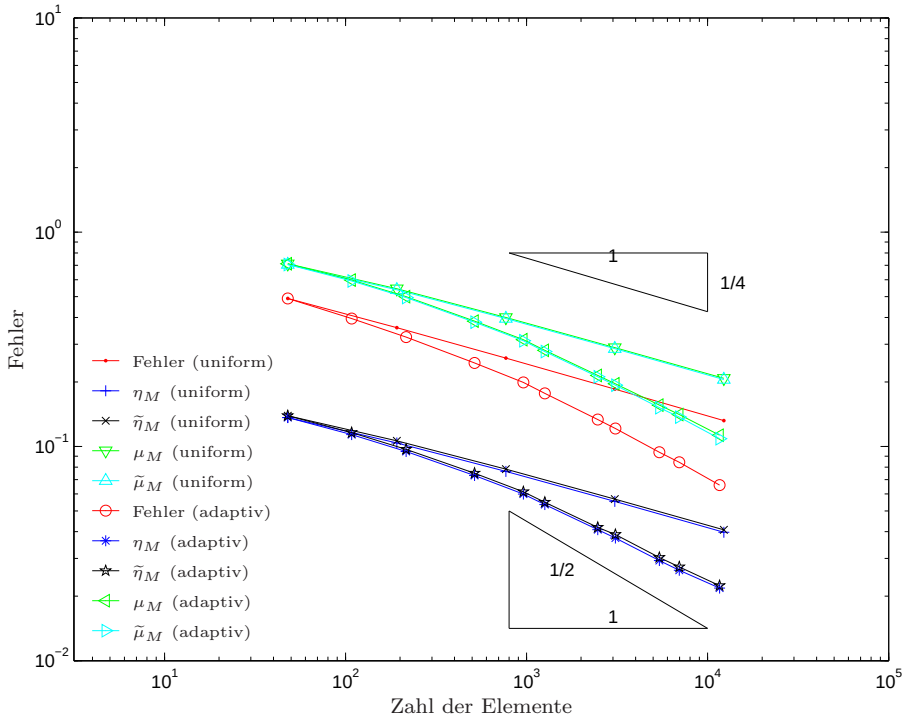


Abbildung 5.18: Fehler $|||\phi - \phi_{h/2}|||$ und Fehlerschätzer $\eta_M, \widetilde{\eta}_M, \mu_M$ sowie $\widetilde{\mu}_M$ in Beispiel 5.3 auf einem L-förmigen Schirm für uniforme und $\widetilde{\mu}_M$-adaptive isotrope Verfeinerungen.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
48	0.491	–	0.137	0.139	0.712	0.706
108	0.396	0.266	0.114	0.116	0.598	0.592
216	0.324	0.289	0.095	0.097	0.501	0.495
516	0.245	0.319	0.073	0.075	0.385	0.380
960	0.199	0.338	0.060	0.061	0.315	0.311
1260	0.177	0.434	0.054	0.055	0.281	0.277
2472	0.134	0.416	0.041	0.042	0.214	0.211
3096	0.121	0.425	0.037	0.039	0.196	0.193
5412	0.094	0.457	0.029	0.030	0.156	0.152
6972	0.084	0.427	0.026	0.027	0.141	0.137
11640	0.066	0.479	0.022	0.022	0.113	0.109

Tabelle 5.17: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_M$-adaptive isotrope Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

hat. In Tabelle 5.19 sind die Verhältnisse

$$\widetilde{\eta}_H/\eta_H \quad \text{und} \quad \mu_H/\widetilde{\mu}_H$$

für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Verfeinerungen angeführt. Wir beobachten, dass die Abweichung zwischen η_H und $\widetilde{\eta}_H$ im uniformen und isotropen Fall verhältnismäßig etwa 0.03 beträgt. Im anisotropen Fall ist eine Zunahme der Differenz zu beobachten. Mit größer werdender Elementzahl im Netz steigt der Quotient von anfänglich 0.03 auf letzten Endes etwa 0.11 an. Ähnlich verhält es sich mit μ_H und $\widetilde{\mu}_H$, wo im uniformen und isotropen Fall eine Abweichung von etwa $0.015 - 0.02$ zu beobachten ist. Im anisotropen Fall nimmt auch hier der Abstand zu, sodass eine Gesamtabweichung von bis zu 0.033 zu sehen ist. Die Abweichung zwischen η_M und $\widetilde{\eta}_M$ beträgt im uniformen Fall etwa 0.025. im isotropen Fall vergrößert sich der Quotient auf bis zu 0.036. Im anisotropen Fall bleibt das Verhältnis bemerkenswerterweise wie im uniformen Fall bei 0.025. Die Abweichung zwischen μ_H und $\widetilde{\mu}_H$ verhält sich ähnlich, im uniformen und anisotropen Fall beobachten wir einen Unterschied von etwa 0.012. Wir halten jedoch fest, dass im letzten Schritt der anisotropen Berechnung ein kleiner Sprung auf etwa 0.027 bemerkbar ist. Im isotropen Fall nimmt die Abweichung auf bis zu 0.038 zu.

In Abbildung 5.20 ist ein Ausschnitt der Folge $\mathcal{T}_h^{(\ell)}$ zu sehen, die aus $\widetilde{\mu}_H$-adaptiver Verfeinerung hervorgeht. Links sind isotrope Netze zu sehen, rechts anisotrope. In Abbildung 5.21 ist ein entsprechender Ausschnitt der Netzfolge $\mathcal{T}_{h/2}^{(\ell)}$ zu sehen, die aus $\widetilde{\mu}_M$-adaptiver Verfeinerung hervorgeht. Auch hier sind links isotrope und rechts anisotrope Netze zu sehen. Wir erkennen jeweils Verfeinerungen zu den Kanten und zu den konvexen Ecken des Schirms hin. Die Verfeinerungen zur inneren Ecke mit den Koordinaten $(1, 1, 0)$ sind notwendig, um die K-Netzeigenschaft zu gewährleisten.

N	$\|\|\|\phi - \phi_{h/2}\|\|\|$	α	η_M	$\widetilde{\eta}_M$	μ_M	$\widetilde{\mu}_M$
48	4.91×10^{-1}	–	1.36×10^{-1}	1.39×10^{-1}	7.12×10^{-1}	7.06×10^{-1}
108	3.96×10^{-1}	0.266	1.14×10^{-1}	1.16×10^{-1}	5.98×1^{-1}	5.92×10^{-1}
184	3.24×10^{-1}	0.376	9.48×10^{-2}	9.70×10^{-2}	5.01×10^{-1}	4.95×10^{-1}
300	2.51×10^{-1}	0.520	7.53×10^{-2}	7.72×10^{-2}	3.94×10^{-1}	3.89×10^{-1}
376	2.07×10^{-1}	0.857	6.30×10^{-2}	6.45×10^{-2}	3.28×10^{-1}	3.24×10^{-1}
552	1.55×10^{-1}	0.753	4.85×10^{-2}	4.97×10^{-2}	2.47×10^{-1}	2.44×10^{-1}
692	1.25×10^{-1}	0.959	3.88×10^{-2}	3.98×10^{-2}	2.02×10^{-1}	1.99×10^{-1}
896	9.37×10^{-2}	1.108	3.02×10^{-2}	3.10×10^{-2}	1.54×10^{-1}	1.52×10^{-1}
1184	6.87×10^{-2}	1.113	2.31×10^{-2}	2.34×10^{-2}	1.16×10^{-1}	1.15×10^{-1}
1376	5.33×10^{-2}	1.685	1.91×10^{-2}	1.95×10^{-2}	9.51×10^{-2}	9.26×10^{-2}
1712	3.98×10^{-2}	1.343	–	–	–	7.39×10^{-2}
2128	3.01×10^{-2}	1.284	–	–	–	5.75×10^{-2}
2640	2.29×10^{-2}	1.275	–	–	–	4.53×10^{-2}
3244	1.73×10^{-2}	1.355	–	–	–	3.60×10^{-2}
3968	1.35×10^{-2}	1.233	–	–	–	2.89×10^{-2}
4816	1.05×10^{-2}	1.275	–	–	–	2.34×10^{-2}
5776	8.59×10^{-3}	1.125	–	–	–	1.94×10^{-2}
7360	6.65×10^{-3}	1.053	–	–	–	1.60×10^{-2}

Tabelle 5.18: Fehler $\|\|\|\phi - \phi_{h/2}\|\|\|$ und Fehlerschätzer η_M, $\widetilde{\eta}_M$, μ_M, $\widetilde{\mu}_M$ sowie experimentelle Konvergenzrate α des Fehlers für eine $\widetilde{\mu}_M$-adaptive anisotrope Netzfolge in Beispiel 5.3. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.

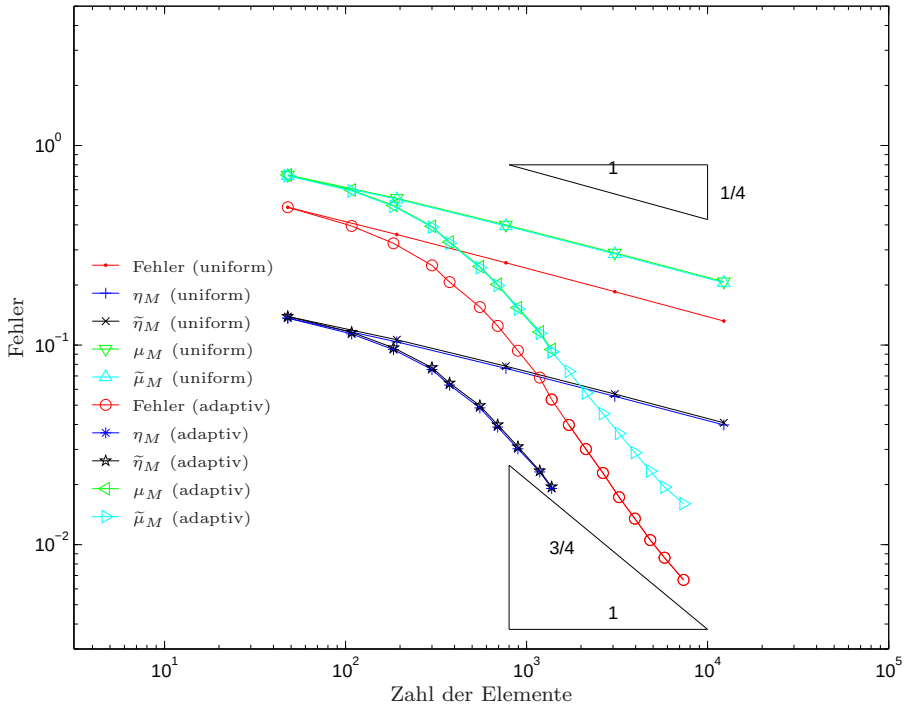


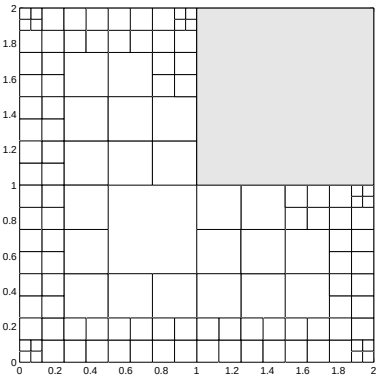
Abbildung 5.19: Fehler $\|\|\|\phi - \phi_h\|\|\|$ und Fehlerschätzer $\eta_M, \widetilde{\eta}_M, \mu_M, \widetilde{\mu}_M$ in Beispiel 5.3 für uniforme und $\widetilde{\mu}_M$-adaptive anisotrope Netzverfeinerung.

uniform			isotrop			anisotrop		
N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$	N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$	N	$\widetilde{\eta}_H/\eta_H$	$\mu_H/\widetilde{\mu}_H$
12	1.028	1.007	12	1.028	1.007	12	1.028	1.007
48	1.036	1.013	17	1.042	1.017	27	1.042	1.017
192	1.033	1.014	54	1.052	1.024	46	1.052	1.024
768	1.032	1.014	129	1.038	1.018	75	1.039	1.019
3072	1.032	1.014	240	1.036	1.017	94	1.037	1.019
			315	1.034	1.016	138	1.033	1.017
			618	1.031	1.015	173	1.035	1.018
			774	1.029	1.014	224	1.034	1.016
			1353	1.029	1.014	296	1.034	1.015
			1743	1.028	1.014	344	1.041	1.016
			2913	1.028	1.013	428	1.046	1.014
			3870	1.028	1.013	532	1.043	1.013
						660	1.043	1.013
						811	1.044	1.013
						992	1.050	1.014
						1205	1.055	1.015
						1442	1.062	1.018
						1860	1.073	1.021
						2251	1.098	1.027
						2846	1.117	1.033

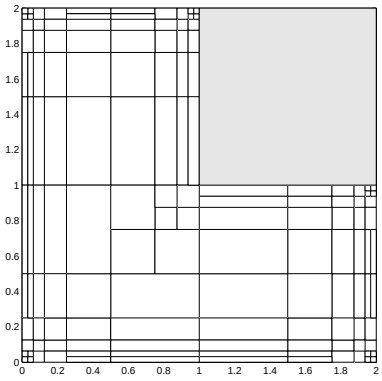
Tabelle 5.19: Verhältnisse $\widetilde{\eta}_H/\eta_H$ und $\mu_H/\widetilde{\mu}_H$ in Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzverfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_h$.

uniform			isotrop			anisotrop		
N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$	N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$	N	$\widetilde{\eta}_M/\eta_M$	$\mu_M/\widetilde{\mu}_M$
48	1.018	1.008	48	1.018	1.008	48	1.018	1.008
192	1.024	1.011	108	1.020	1.009	108	1.020	1.009
768	1.025	1.012	216	1.023	1.011	184	1.023	1.011
3072	1.026	1.012	516	1.024	1.012	300	1.024	1.012
12288	1.026	1.012	960	1.025	1.012	376	1.024	1.011
			1260	1.025	1.013	552	1.026	1.012
			2472	1.025	1.014	692	1.025	1.012
			3096	1.038	1.014	896	1.026	1.016
			5412	1.034	1.027	1184	1.012	1.016
			6072	1.036	1.028	1376	1.020	1.027
			11640	1.027	1.038			

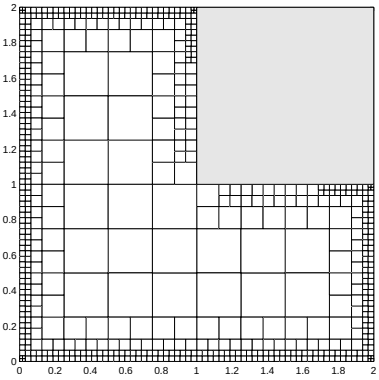
Tabelle 5.20: Verhältnisse $\widetilde{\eta}_H/\eta_H$ und $\mu_H/\widetilde{\mu}_H$ in Beispiel 5.3 für uniforme und $\widetilde{\mu}_H$-adaptive isotrope bzw. anisotrope Netzverfeinerung. Es bezeichnet N die Anzahl der Elemente in $\mathcal{T}_{h/2}$.



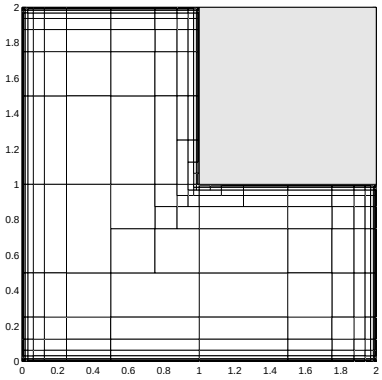
$\mathcal{T}_h^{(3)}, N = 129$



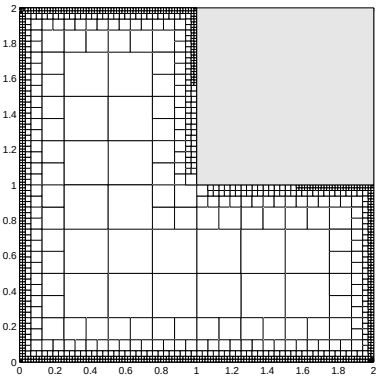
$\mathcal{T}_h^{(5)}, N = 138$



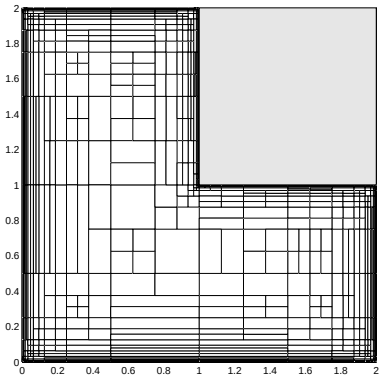
$\mathcal{T}_h^{(6)}, N = 618$



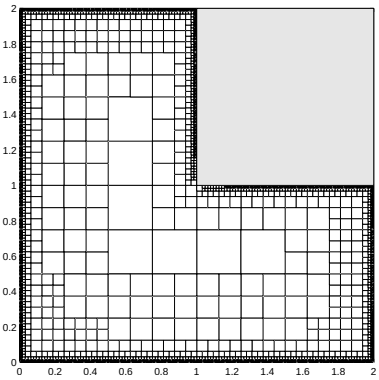
$\mathcal{T}_h^{(8)}, N = 428$



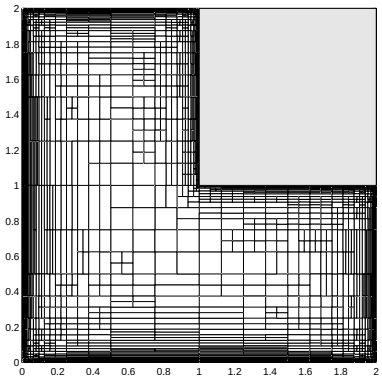
$\mathcal{T}_h^{(8)}, N = 1353$



$\mathcal{T}_h^{(15)}, N = 1205$

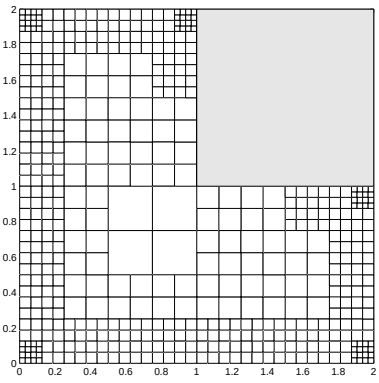


$\mathcal{T}_h^{(11)}, N = 3870$

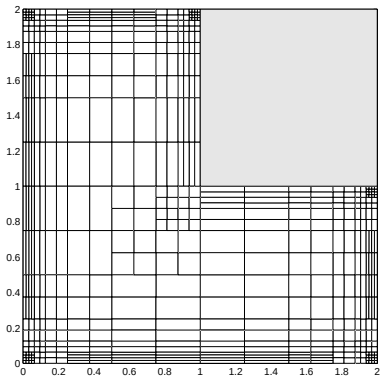


$\mathcal{T}_h^{(19)}, N = 2846$

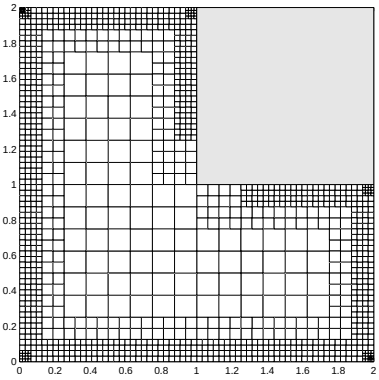
Abbildung 5.20: Ausschnitt der Folge von Netzen, die durch $\widetilde{\mu}_H$-adaptive isotrope (links) bzw. anisotrope (rechts) Verfeinerungen in Beispiel 5.3 hervorgeht.



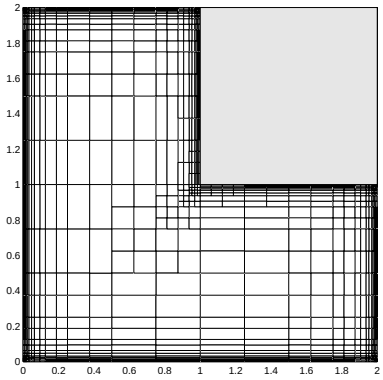
$\mathcal{T}_{h/2}^{(3)}, N = 516$



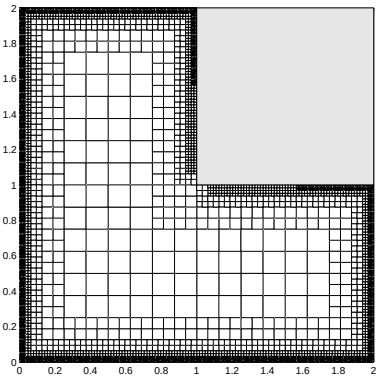
$\mathcal{T}_{h/2}^{(5)}, N = 552$



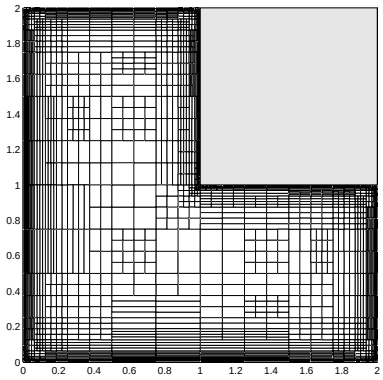
$\mathcal{T}_{h/2}^{(5)}, N = 1260$



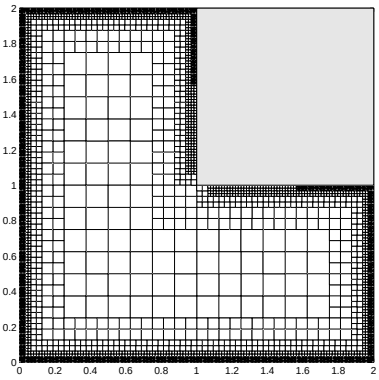
$\mathcal{T}_{h/2}^{(8)}, N = 1184$



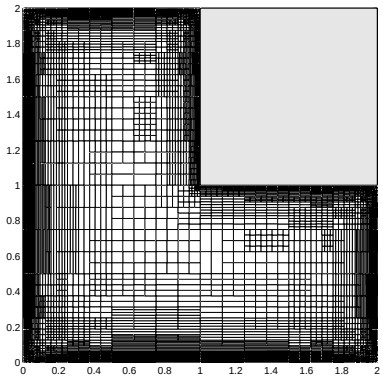
$\mathcal{T}_{h/2}^{(8)}, N = 5412$



$\mathcal{T}_{h/2}^{(14)}, N = 3968$



$\mathcal{T}_{h/2}^{(10)}, N = 11640$



$\mathcal{T}_{h/2}^{(17)}, N = 7360$

Abbildung 5.21: Ausschnitt der Folge von Netzen, die durch $\widetilde{\mu}_M$-adaptive isotrope (links) bzw. anisotrope (rechts) Verfeinerungen in Beispiel 5.3 hervorgeht.

Anhang A

Quelltexte

A.1 Das Programm *symm_mu_H_estimator*

```
1  //************************************************************************//
2  //File:               symm_mu_H_estimator.cpp                             //
3  //This file implements the software used for numerical experiment         //
4  //using eta_H, eta_H_tilde, mu_H and mu_H_tilde.                          //
5  //************************************************************************//
6  //Author: Samuel Ferraz-Leite                                             //
7  //Date: 07.2007                                                           //
8  //Version 1.0 stable                                                      //
9  //************************************************************************//
10
11 //includes of C-standardlibraries
12 #include <ctype.h>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <math.h>
16 #include <sys/times.h>
17 #include <string.h>
18 #include <time.h>
19
20
21 //Includes from the FLBEM
22 #include "triangulation.h"
23 #include "triangulationio.h"
24 #include "usual_meshes.h"
25 #include "quadrature.h"
26 #include "geometrics.h"
27 #include "integral_operators.h"
28
29 //includes of C++-standardlibraries
30 #include <iostream>
31 #include <fstream>
32 #include <map>
33
34 //Definition of constants for filenames of logfiles,
35 //convergencerate, energynormsquare etc...
36 #define FILENAME "L_isotrope.log"
37 #define MESHFILE "coarse_mesh_mu_H.ps"
38 #define ENORMSQUARE 8.284570593849
39 #define KONVERGENCERATE 2./4.
40
41 //ENROMSQUARE box=16.60470319300621
42 //ENORMSQUARE L2=8.284570593849
43
44 using namespace triangulation;
```

```
45  using namespace geometrics;
46  using namespace quadrature;
47  using namespace BEM;
48
49  //L^2-Projection from \P^0(\T_{h/2}) to \P^0(\T_h)
50  std::vector<double> L2_projection_TH_Th(double* fine_mesh_coeff)
51  {
52    std::vector<double> return_var=std::vector<double>(4);
53    return_var[0]=(25*fine_mesh_coeff[0]-5*fine_mesh_coeff[1]+fine_mesh_coeff[2]-
54        5*fine_mesh_coeff[3])/16.;
55    return_var[1]=(-5*fine_mesh_coeff[0]+25*fine_mesh_coeff[1]-5*fine_mesh_coeff[2]+
56        fine_mesh_coeff[3])/16.;
57    return_var[2]=(fine_mesh_coeff[0]-5*fine_mesh_coeff[1]+25*fine_mesh_coeff[2]-
58        5*fine_mesh_coeff[3])/16.;
59    return_var[3]=(-5*fine_mesh_coeff[0]+fine_mesh_coeff[1]-5*fine_mesh_coeff[2]+
60        25*fine_mesh_coeff[3])/16.;
61    return return_var;
62  }
63
64  int main()
65  {
66    //*********************************************//
67    //STARTUP AND INITIALISATIONS                  //
68    //*********************************************//
69    double theta=0.5;
70    double tau=0.0;
71
72    std::ofstream expfile(FILENAME);
73    expfile<<"%NDoF(fine)␣\t␣|||u_h|||^2␣\t␣|||u-u_h|||␣\t␣|||u_H|||␣\t␣|||u-u_H|||␣\t
          \t␣mu_hmin␣\t\t␣eta_H␣\t\t␣mu_hmin␣t␣\t␣eta_H_tilde\n";
74    expfile.close();
75
76    std::cout<<"This␣is␣an␣experiment␣for␣error-estimators␣using␣rectangular␣elements\
          n"
77        <<"and␣analytical␣computation␣of␣the␣galerkin␣elements\n\n";
78    std::cout<<"Creating␣mesh\n";
79
80    //Change this line to use a different surface
81    Triangulation3D large_mesh=create_L2D_mesh(RECTANGLE,true);
82
83    large_mesh.refine_uniformily();
84
85    //*********************************************//
86    //ITERATIVE COMPUTATION USING ADAPTIVE ALGORITHM//
87    //*********************************************//
88    while(large_mesh.size()<2700) //use your machines memory limit
89    {
90      //-------------------------------------------//
91      //MANAGE MESHES                              //
92      //-------------------------------------------//
93      clock_t t1,t2;
94      std::cout<<"Created␣coarse␣mesh␣with␣"<<large_mesh.size()<<"␣rectangles.\n\n";
95      std::cout<<"Triangulation3D␣fine_mesh=large_mesh;\n";
96      Triangulation3D fine_mesh=Triangulation3D(large_mesh);
97      std::cout<<"Done!\n\n";
98      std::map<int,int*> large_fine=std::map<int,int*>();
99      fine_mesh.reserve(4*large_mesh.size());
100     for(int i=0;i<large_mesh.size();++i)
101     {
102       fine_mesh[i]->mark_red(false);
103       std::vector<Shape*> new_rectangles=fine_mesh[i]->split();
104       int* new_indizes=(int*)malloc(4*sizeof(int));
105       new_indizes[0]=i;
```

```
106         fine_mesh.push_back(new_rectangles[0]);
107         new_indizes[1]=fine_mesh.size()-1;
108         fine_mesh.push_back(new_rectangles[1]);
109         new_indizes[2]=fine_mesh.size()-1;
110         fine_mesh.push_back(new_rectangles[2]);
111         new_indizes[3]=fine_mesh.size()-1;
112         large_fine.insert(std::pair<int,int*>(i,new_indizes));
113       }
114
115       //-----------------------------------------//
116       //BUILD H-MATRIX                           //
117       //-----------------------------------------//
118       std::cout<<"Building␣H-Matrix\n\n";
119       t1=clock();
120       SLP A;
121       A.build(fine_mesh,CONSTANT,ACA_EPS/pow((double)fine_mesh.size(),KONVERGENCERATE)
              );
122       t2=clock();
123       outputrank_supermatrix(A.A,"Ark.ps");
124       std::cout<<"Storage␣requirements␣for␣Hmatrix\n";
125       size_t size_A=getsize_supermatrix(A.A);
126       std::cout<<"Fill:␣"<<double(t2-t1)/CLOCKS_PER_SEC<<"␣sec.\n";
127       std::cout<<"size="<<size_A/1024./1024.<<"MB␣["<<size_A/1024./fine_mesh.size()<<"
              KB/N]\n\n";
128
129
130       //-----------------------------------------//
131       //BUILD H-MATRIX FOR OTHER ESTIMATOR       //
132       //-----------------------------------------//
133       std::cout<<"Building␣coarse-mesh␣H-matrix\n\n";
134       t1=clock();
135       SLP B;
136       B.build(large_mesh,large_mesh,CONSTANT,ACA_EPS/pow((double)large_mesh.size(),
              KONVERGENCERATE));
137       t2=clock();
138       std::cout<<"Storage␣requirements␣for␣Hmatrix\n";
139       size_A=getsize_supermatrix(B.A);
140       outputrank_supermatrix(B.A,"Brk.ps");
141       std::cout<<"Fill:␣"<<double(t2-t1)/CLOCKS_PER_SEC<<"␣sec.\n";
142       std::cout<<"size="<<size_A/1024./1024.<<"MB\n\n";
143
144       //-----------------------------------------//
145       //SOLVE SYSTEM TO OBTAIN SOLUTION ON FINE MESH//
146       //-----------------------------------------//
147       std::cout<<"Solving␣Symms␣IE\n";
148       double* b=(double*)malloc(fine_mesh.size()*sizeof(double));
149       for(int i=0;i<fine_mesh.size();++i)
150         b[i]=fine_mesh[i]->gram_det();
151       double* uh=A/b;
152       double* Pi0_uh=(double*)malloc(large_mesh.size()*sizeof(double));
153
154       //-------------------------------------------------------------//
155       //COMPUTE THE ERROR IN THE ENERGY-NORM OF THE FINE-MESH SOLUTION//
156       //-------------------------------------------------------------//
157       double enorm=0.;
158       for(int i=0;i<fine_mesh.size();++i)
159         enorm+=uh[i]*b[i];
160       std::cout<<"The␣energy␣norm␣of␣the␣solution␣is␣"<<sqrt(enorm)<<"\n"
161           <<"The␣error␣in␣the␣energy␣norm␣is␣"<<sqrt(ENORMSQUARE-enorm)<<"\n\n";
162
163       //-----------------------------------------//
164       //SOLVE COARSE SYSTEM FOR ETA_H ESTIMATOR  //
165       //-----------------------------------------//
```

```
166     std::cout<<"Solving␣coarse␣system\n";
167     double* d=(double*)malloc(large_mesh.size()*sizeof(double));
168     for(int i=0;i<large_mesh.size();++i)
169       d[i]=large_mesh[i]->gram_det();
170     double* uH=B/d;
171 
172     //-----------------------------------------------------------------//
173     //COMPUTE THE ERROR IN THE ENERGY-NORM OF THE COARSE-MESH SOLUTION//
174     //-----------------------------------------------------------------//
175     double enorm_large=0.;
176     for(int i=0;i<large_mesh.size();++i)
177       enorm_large+=uH[i]*d[i];
178     std::cout<<"\nThe␣energy␣norm␣of␣the␣coarse␣solution␣is␣"<<sqrt(enorm_large)<<"\
          n"
179           <<"The␣error␣in␣the␣energy␣norm␣is␣"<<sqrt(ENORMSQUARE-enorm_large)<<"\
              n\n";
180 
181 
182     //---------------------------------------------//
183     //COMPUTE THE ERROR ESTIMATORS                 //
184     //---------------------------------------------//
185     std::cout<<"Starting␣computation␣of␣the␣error-estimators\n\n";
186     t1=clock();
187 
188     //Compute the error-vector uH-uh
189     double* x_error=(double*)malloc(fine_mesh.size()*sizeof(double));
190     for(int l=0;l<large_mesh.size();++l)
191       for(int j=0;j<4;++j)
192         x_error[(large_fine[l])[j]]=uH[l]-uh[(large_fine[l])[j]];
193     //error contains the local error-squares.
194     std::vector<double> error_mu_H_hmin=std::vector<double>(large_mesh.size());
195     std::vector<double> error_mu_H_tilde_hmin=std::vector<double>(large_mesh.size())
          ;
196     double max_error=0.;
197     double mu_H_hmin=0.;
198     double mu_H_tilde_hmin=0.;
199     for(int l=0;l<large_mesh.size();++l)
200     {
201       double* fine_mesh_coeff=(double*)malloc(4*sizeof(double));
202       for(int j=0;j<4;++j)
203         fine_mesh_coeff[j]=uh[(large_fine[l])[j]];
204       Pi0_uh[l]=0.25*(fine_mesh_coeff[0]+fine_mesh_coeff[1]+
205                       fine_mesh_coeff[2]+fine_mesh_coeff[3]);
206 
207       //Computations for mu_H_hmin
208       double k=x_error[(large_fine[l])[0]];
209       double true_error0=k*k/4.;
210       k=x_error[(large_fine[l])[1]];
211       double true_error1=k*k/4.;
212       k=x_error[(large_fine[l])[2]];
213       double true_error2=k*k/4.;
214       k=x_error[(large_fine[l])[3]];
215       double true_error3=k*k/4.;
216 
217       //Computations for mu_H_tilde
218       k=fine_mesh_coeff[0];
219       double a=Pi0_uh[l];
220       double error0=(a-k)*(a-k)/4.;
221       k=fine_mesh_coeff[1];
222       double error1=(a-k)*(a-k)/4.;
223       k=fine_mesh_coeff[2];
224       double error2=(a-k)*(a-k)/4.;
225       k=fine_mesh_coeff[3];
```

```
226       double error3=(a-k)*(a-k)/4.;
227
228       std::vector<Point3D> vertices=large_mesh[l]->get_vertices();
229       double diameter=vertices[0].distance(vertices[1]);
230       if(diameter<vertices[1].distance(vertices[2]))
231         diameter=vertices[1].distance(vertices[2]);
232       double hmin=vertices[0].distance(vertices[1]);
233       if(hmin>vertices[1].distance(vertices[2]))
234         hmin=vertices[1].distance(vertices[2]);
235       error_mu_H_hmin[l]=(true_error0+true_error1+true_error2+true_error3)*
236                          ((large_mesh[l]->gram_det()));
237       error_mu_H_tilde_hmin[l]=(error0+error1+error2+error3)*((large_mesh[l]->
              gram_det()));
238       error_mu_H_hmin[l]*=hmin;
239       error_mu_H_tilde_hmin[l]*=hmin;
240       mu_H_hmin+=error_mu_H_hmin[l];
241       mu_H_tilde_hmin+=error_mu_H_tilde_hmin[l];
242       if (max_error<error_mu_H_tilde_hmin[l])
243         max_error=error_mu_H_tilde_hmin[l];
244       free(fine_mesh_coeff);
245     }
246     mu_H_hmin=sqrt(mu_H_hmin);
247     mu_H_tilde_hmin=sqrt(mu_H_tilde_hmin);
248     t2=clock();
249     std::cout<<"The␣maximal␣estimated␣error␣on␣one␣element␣is␣"<<max_error<<"\n";
250     std::cout<<"mu_H_hmin=␣"<<mu_H_hmin<<"\n";
251     std::cout<<"Error-indicator␣computed.␣Operation␣took␣"
252              <<double((t2-t1)/CLOCKS_PER_SEC)<<"␣sec."
253              <<"\n\n";
254
255     std::cout<<"Computing␣estimator␣eta_H_tilde\n";
256     double eta_H_tilde=0.;
257     for(int l=0;l<large_mesh.size();++l)
258       for(int j=0;j<4;++j)
259         x_error[(large_fine[l])[j]]=uh[(large_fine[l])[j]]-Pi0_uh[l];
260     double* c=A*x_error;
261     for(int i=0;i<fine_mesh.size();++i)
262       eta_H_tilde+=c[i]*x_error[i];
263     eta_H_tilde=sqrt(eta_H_tilde);
264     std::cout<<"Done!␣eta_H_tilde="<<eta_H_tilde<<"\n";
265
266     std::cout<<"Computing␣true␣estimator␣eta_H\n";
267     double eta_H=0.;
268     for(int l=0;l<large_mesh.size();++l)
269       for(int j=0;j<4;++j)
270         x_error[(large_fine[l])[j]]=uH[l]-uh[(large_fine[l])[j]];
271     free(c);
272     c=A*x_error;
273     for(int i=0;i<fine_mesh.size();++i)
274       eta_H+=c[i]*x_error[i];
275     eta_H=sqrt(eta_H);
276     std::cout<<"Done!␣true␣eta_H="<<eta_H<<"\n";
277
278
279     //-------------------------------------//
280     //REFINE THE COARSE MESH               //
281     //-------------------------------------//
282     std::cout<<"Refining␣the␣coarse␣mesh\n";
283     t1=clock();
284     for(int l=0;l<large_mesh.size();++l)
285     {
286       std::vector<double> fine_mesh_coeff=std::vector<double>(4);
287       for(int j=0;j<4;++j)
```

```
288         fine_mesh_coeff[j]=uh[(large_fine[l])[j]];
289       double b=-0.25*fine_mesh_coeff[0]-0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]+
290           0.25*fine_mesh_coeff[3];
291       double c=-0.25*fine_mesh_coeff[0]+0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]-
292           0.25*fine_mesh_coeff[3];
293       double d=0.25*fine_mesh_coeff[0]-0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]-
294           0.25*fine_mesh_coeff[3];
295       if(error_mu_H_tilde_hmin[l]>=theta*max_error)
296       {
297         if(fabs(tau*b)>=sqrt(fabs(c)*fabs(c)+fabs(d)*fabs(d)))
298           large_mesh[l]->mark_red_y();
299         else if(fabs(tau*c)>=sqrt(fabs(b)*fabs(b)+fabs(d)*fabs(d)))
300           large_mesh[l]->mark_red_x();
301         else
302           large_mesh[l]->mark_red();
303       }
304     }
305     large_mesh.refine();
306     large_mesh.regulate_diameters(4);
307     t2=clock();
308     std::cout<<"Mesh␣was␣refined.␣Operation␣took␣"
309              <<double((t2-t1)/CLOCKS_PER_SEC)<<"␣sec.\n\n";
310
311     //-----------------------------------//
312     //WRITE RESULTS TO A LOG-FILE        //
313     //-----------------------------------//
314     expfile.open(FILENAME,std::ios::app);
315     expfile.precision(12);
316     expfile<<fine_mesh.size()<<"\t\t"<<fabs(enorm)
317            <<"\t␣"<<sqrt(ENORMSQUARE-enorm)<<"\t"
318            <<fabs(enorm_large)<<"\t"<<sqrt(ENORMSQUARE-enorm_large)
319            <<"\t"<<mu_H_hmin<<"\t"<<eta_H<<"\t\t"
320            <<mu_H_tilde_hmin<<"\t"<<eta_H_tilde<<"\n";
321     expfile.close();
322
323     //-----------------------------------//
324     //FREE ALLOCATED MEMORY              //
325     //-----------------------------------//
326     free(uH);
327     free(b);
328     free(c);
329     free(uh);
330     free(x_error);
331   }
332   std::cout<<"Thank␣you␣for␣using␣FLBEM!\n";
333 }
```

A.2 Das Programm *symm_mu_M_estimator*

```
1  //************************************************************************//
2  //File:               symm_mu_M_estimator.cpp                             //
3  //This file implements the software used for numerical experiments        //
4  //using eta_M, eta_M_tilde, mu_M and mu_M_tilde.                          //
5  //************************************************************************//
6  //Author: Samuel Ferraz-Leite                                             //
7  //Date: 07.2007                                                           //
8  //Version 1.0 stable                                                      //
9  //************************************************************************//
10
11 //includes of C-standardlibraries
```

```
12 #include <ctype.h>
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <math.h>
16 #include <sys/times.h>
17 #include <string.h>
18 #include <time.h>
19
20
21 //Includes from the bemlib for the adaptive mesh
22 #include "triangulation.h"
23 #include "triangulationio.h"
24 #include "usual_meshes.h"
25 #include "quadrature.h"
26 #include "geometrics.h"
27 #include "integral_operators.h"
28
29 //Includes from the standardlibraries
30 #include <iostream>
31 #include <fstream>
32 #include <map>
33
34 #define FILENAME "box_isotrope.log"
35 #define MESHFILE "box_isotrope.ps"
36 #define ENORMSQUARE 16.60470319300621
37 #define KONVERGENCERATE 3./4.
38 #define PI acos(-1)
39
40 //ENROMSQUARE box=16.60470319300621
41 //ENORMSQUARE L2=8.284570593849
42
43 using namespace triangulation;
44 using namespace geometrics;
45 using namespace quadrature;
46 using namespace BEM;
47
48 std::vector<double> L2_projection_TH_Th(double* fine_mesh_coeff)
49 {
50   std::vector<double> return_var=std::vector<double>(4);
51   return_var[0]=(25*fine_mesh_coeff[0]-5*fine_mesh_coeff[1]+fine_mesh_coeff[2]-
52       5*fine_mesh_coeff[3])/16.;
53   return_var[1]=(-5*fine_mesh_coeff[0]+25*fine_mesh_coeff[1]-5*fine_mesh_coeff[2]+
54       fine_mesh_coeff[3])/16.;
55   return_var[2]=(fine_mesh_coeff[0]-5*fine_mesh_coeff[1]+25*fine_mesh_coeff[2]-
56       5*fine_mesh_coeff[3])/16.;
57   return_var[3]=(-5*fine_mesh_coeff[0]+fine_mesh_coeff[1]-5*fine_mesh_coeff[2]+
58       25*fine_mesh_coeff[3])/16.;
59   return return_var;
60 }
61
62 int main()
63 {
64   //*******************************************//
65   //STARTUP AND INITIALISATIONS                //
66   //*******************************************//
67   double theta=0.5;
68   double tau=0.5;
69   std::ofstream expfile(FILENAME);
70   expfile<<"%NDoF␣\t␣|||u_h|||^2␣\t␣|||u_H|||^2␣\t␣|||u-u_h|||␣\t␣eta_M"
71          <<"␣\t\t␣eta_pi␣\t␣mu_M␣\t\t␣mu_pi\n";
72   expfile.close();
73   std::cout<<"This␣is␣a␣test␣for␣ACA␣approximation␣using␣rectangular␣elements\n"
74       <<"and␣analytical␣computation␣of␣the␣galerkin␣elements␣with␣gradient␣recovery\
```

```
          n"
75        <<"providing␣many␣estimators␣mu_pi,␣eta_pi␣and␣eta_M.␣For␣details␣see␣
              documentation\n\n";
76    std::cout<<"Creating␣mesh\n";
77
78    Triangulation3D large_mesh=create_box_mesh(RECTANGLE,true);
79    large_mesh.refine_uniformily();
80
81    //*********************************************//
82    //ITERATIVE COMPUTATION USING ADAPTIVE ALGORITHM//
83    //*********************************************//
84    while(large_mesh.size()<2700)
85    {
86      clock_t t1,t2;
87      //-------------------------------------------//
88      //MANAGE MESHES                              //
89      //-------------------------------------------//
90      std::cout<<"Created␣coarse␣mesh␣with␣"<<large_mesh.size()<<"␣rectangles.\n\n";
91      std::cout<<"Triangulation3D␣fine_mesh=large_mesh;\n";
92      Triangulation3D fine_mesh=Triangulation3D(large_mesh);
93      std::cout<<"Done!\n\n";
94      std::map<int,int*> large_fine=std::map<int,int*>();
95      fine_mesh.reserve(4*large_mesh.size());
96      for(int i=0;i<large_mesh.size();++i)
97      {
98        fine_mesh[i]->mark_red(false);
99        std::vector<Shape*> new_rectangles=fine_mesh[i]->split();
100       int* new_indizes=(int*)malloc(4*sizeof(int));
101       new_indizes[0]=i;
102       fine_mesh.push_back(new_rectangles[0]);
103       new_indizes[1]=fine_mesh.size()-1;
104       fine_mesh.push_back(new_rectangles[1]);
105       new_indizes[2]=fine_mesh.size()-1;
106       fine_mesh.push_back(new_rectangles[2]);
107       new_indizes[3]=fine_mesh.size()-1;
108       large_fine.insert(std::pair<int,int*>(i,new_indizes));
109     }
110     draw_tr3d_projection_to_xy("L2_fine.ps",fine_mesh);
111     //------------------------------//
112     //Build the H-Matrices A,B and C//
113     //------------------------------//
114     std::cout<<"Building␣H-Matrix␣A\n\n";
115     t1=clock();
116     SLP A;
117     A.build(fine_mesh,CONSTANT,ACA_EPS/pow(fine_mesh.size(),KONVERGENCERATE));
118     outputrank_supermatrix(A.A,"Ark.ps");
119     t2=clock();
120     std::cout<<"Storage␣requirements␣for␣Hmatrix\n";
121     size_t size_A=getsize_supermatrix(A.A);
122     std::cout<<"Fill:␣"<<double(t2-t1)/CLOCKS_PER_SEC<<"␣sec.,";
123     std::cout<<"size="<<size_A/1024./1024.<<"MB␣["<<size_A/1024./fine_mesh.size()<<"
            KB/N]\n\n";
124
125     std::cout<<"Building␣H-Matrix␣B\n\n";
126     t1=clock();
127     SLP B;
128     B.build(large_mesh,large_mesh,LINEAR,ACA_EPS/pow(large_mesh.size(),
            KONVERGENCERATE));
129     outputrank_supermatrix(B.A,"Brk.ps");
130     t2=clock();
131     std::cout<<"Storage␣requirements␣for␣Hmatrix\n";
132     size_A=getsize_supermatrix(B.A);
133     std::cout<<"Fill:␣"<<double(t2-t1)/CLOCKS_PER_SEC<<"␣sec.,";
```

```
134     std::cout<<"size="<<size_A/1024./1024.<<"MB [("<<size_A/1024./fine_mesh.size()<<"
            KB/N]\n\n";
135
136     std::cout<<"Building H-Matrix C\n\n";
137     t1=clock();
138     SLP C;
139     C.build(large_mesh,fine_mesh,LINCONST,ACA_EPS/pow(large_mesh.size(),
            KONVERGENCERATE));
140     outputrank_supermatrix(C.A,"Crk.ps");
141     t2=clock();
142     std::cout<<"Storage requirements for Hmatrix\n";
143     size_A=getsize_supermatrix(C.A);
144     std::cout<<"Fill: "<<double(t2-t1)/CLOCKS_PER_SEC<<" sec.,";
145     std::cout<<"size="<<size_A/1024./1024.<<"MB\n\n";
146
147     //-----------------------------//
148     //solve the IE for the RHS b==1//
149     //-----------------------------//
150     std::cout<<"Solving Symms IE for RHS b=1\n";
151     double* b=(double*)malloc(fine_mesh.size()*sizeof(double));
152     for(int i=0;i<fine_mesh.size();++i)
153       b[i]=fine_mesh[i]->gram_det();
154     double* x_large=(double*)malloc(4*large_mesh.size()*sizeof(double));//Each
            element has
155     double* x=A/b;
156
157     //-------------------------------------------------//
158     //Compute the energy norm of the discrete solution x//
159     //-------------------------------------------------//
160     double enorm=0.;
161     for(int i=0;i<fine_mesh.size();++i)
162       enorm+=b[i]*x[i];
163     std::cout<<"|||u_h|||^2="<<fabs(enorm)<<" |||u_h|||="<<sqrt(fabs(enorm))<<"\n"
164         <<"|||u-u_h|||="<<sqrt(fabs(ENORMSQUARE-fabs(enorm)))<<"\n\n";
165
166     //-----------------------------------------------//
167     //Compute the error estimator mu_pi and Pi_H*u_h//
168     //-----------------------------------------------//
169     std::cout<<"Starting computation of the error by gradient recovery\n\n";
170     t1=clock();
171     //error contains the local error-squares.
172     std::vector<double> error=std::vector<double>(large_mesh.size());
173     double max_error=0.;
174     double total_error=0.;
175     for(int l=0;l<large_mesh.size();++l)
176     {
177       double* fine_mesh_coeff=(double*)malloc(4*sizeof(double));
178       for(int j=0;j<4;++j)
179         fine_mesh_coeff[j]=x[(large_fine[l])[j]];
180       std::vector<double> x_large_new=L2_projection_TH_Th(fine_mesh_coeff);//
              representation
181                                                                   //on the reference
                                                                        element
182       for(int j=0;j<4;++j)
183         x_large[4*l+j]=x_large_new[j];
184       double k=fine_mesh_coeff[0];
185       double a=x_large_new[0];
186       double b=x_large_new[1];
187       double c=x_large_new[2];
188       double d=x_large_new[3];
189       double error0=0.08506944444*a*a+0.04861111111*a*b+0.04861111111*a*d-
190                     0.28125*k*a+0.01388888889*a*c+0.006944444444*b*c+
191                     0.001736111111*c*c+0.25*k*k-0.09374999999*k*b-0.03125*c*k+
```

```
192                   0.006944444444*c*d-0.09374999999*d*k+0.013888888889*b*d+
193                   0.012152777778*b*b+0.01215277778*d*d;
194       k=fine_mesh_coeff[1];
195       double error1=0.01215277778*a*a+0.04861111111*a*b+0.006944444444*a*d
              -0.09374999999*k*a+
196                   0.01388888889*a*c+0.01215277778*c*c+0.250000000000*k*k
                          +0.04861111111*b*c-
197                   0.09374999999*c*k-0.03125000000*d*k+0.013888888889*b*d
                          -0.28125000000*k*b+
198                   0.08506944444*b*b+0.00173611111*d*d+0.006944444444*c*d;
199       k=fine_mesh_coeff[2];
200       double error2=0.25000000000*k*k-0.09374999999*k*b+0.001736111111*a*a
              +0.00694444444*a*b+
201                   0.01215277778*b*b-0.03125000000*k*a+0.085069444444*c*c
                          +0.01215277778*d*d+
202                   0.04861111111*c*d+0.01388888889*a*c+0.006944444444*a*d
                          +0.04861111111*b*c+
203                   0.01388888889*b*d-0.28125000000*c*k-0.093749999999*d*k;
204       k=fine_mesh_coeff[3];
205       double error3=0.25000000000*k*k-0.03125000000*k*b+0.012152777778*a*a
              +0.00694444444*a*b+
206                   0.00173611111*b*b-0.09374999999*k*a+0.012152777778*c*c
                          +0.08506944444*d*d+
207                   0.04861111111*c*d+0.01388888889*a*c+0.048611111111*a*d
                          +0.00694444444*b*c+
208         0.01388888889*b*d-0.09374999999*c*k-0.281250000000*d*k;
209       std::vector<Point3D> vertices=large_mesh[l]->get_vertices();
210       double diameter=vertices[0].distance(vertices[2]);
211       if(diameter<vertices[1].distance(vertices[3]))
212         diameter=vertices[1].distance(vertices[3]);
213       double hmin=vertices[0].distance(vertices[1]);
214       if(hmin>vertices[1].distance(vertices[2]))
215         hmin=vertices[1].distance(vertices[2]);
216       error[l]=(error0+error1+error2+error3)*(large_mesh[l]->gram_det())*hmin;
217       total_error+=error[l];
218       if (max_error<error[l])
219         max_error=error[l];
220       free(fine_mesh_coeff);
221     }
222     total_error=sqrt(total_error);
223     t2=clock();
224     std::cout<<"The maximal estimated error on one element is "<<max_error<<"\n";
225     std::cout<<"mu_pi="<<total_error<<"\n";
226     std::cout<<"Error-indicator mu_pi computed. Operation took "
227              <<double((t2-t1)/CLOCKS_PER_SEC)<<" sec."<<"\n\n";
228
229     //------------------------------//
230     //Compute the enrgy-norm of u_H//
231     //------------------------------//
232     std::cout<<"starting computation of |||u_H|||\n";
233     double* uH=(double*)malloc(4*large_mesh.size()*sizeof(double));
234     for(int i=0;i<4*large_mesh.size();++i)
235       uH[i]=x_large[i];
236     double* BuH=B*uH;
237     double error2=0.;
238     for(int i=0;i<4*large_mesh.size();++i)
239       error2+=BuH[i]*uH[i];
240     std::cout<<"Done! |||uH|||^2="<<error2<<", which means |||uH|||="<<sqrt(fabs(
          error2))<<"\n\n";
241
242     //---------------------------------//
243     //Compute the error-estimator eta_M//
244     //---------------------------------//
```

```
245     std::cout<<"Starting␣computation␣of␣error-indicator␣eta_M\n";
246     double* c=C*x;
247     double* y=B/c;
248     double eta_M=0.;
249     for(int i=0;i<4*large_mesh.size();++i)
250       eta_M+=c[i]*y[i];
251     std::cout<<"|||G_Hu_h|||^2="<<eta_M<<"\n\n";
252     eta_M=fabs(enorm-eta_M);
253     std::cout<<"Error␣estimation␣done...␣eta_M="<<sqrt(eta_M)<<"\n\n";
254
255     //--------------------------------//
256     //Compute the error-estimator mu_M//
257     //--------------------------------//
258     std::cout<<"Starting␣computation␣of␣mu_M\n";
259     double mu_M=0.;
260     double* error_mu_M=(double*)malloc(large_mesh.size()*sizeof(double));
261     for(int l=0;l<large_mesh.size();++l)
262     {
263       double* fine_mesh_coeff=(double*)malloc(4*sizeof(double));
264       for(int j=0;j<4;++j)
265         fine_mesh_coeff[j]=x[(large_fine[l])[j]];
266       double k=fine_mesh_coeff[0];
267       double a=y[4*l+0];
268       double b=y[4*l+1];
269       double c=y[4*l+2];
270       double d=y[4*l+3];
271       double error0=0.08506944444*a*a+0.04861111111*a*b+0.04861111111*a*d-
272                     0.28125*k*a+0.01388888889*a*c+0.006944444444*b*c+
273                     0.001736111111*c*c+0.25*k*k-0.09374999999*k*b-0.03125*c*k+
274                     0.006944444444*c*d-0.09374999999*d*k+0.013888888889*b*d+
275                     0.012152777778*b*b+0.01215277778*d*d;
276       k=fine_mesh_coeff[1];
277       double error1=0.01215277778*a*a+0.04861111111*a*b+0.006944444444*a*d
              -0.09374999999*k*a+
278                     0.01388888889*a*c+0.01215277778*c*c+0.250000000000*k*k
                          +0.04861111111*b*c-
279                     0.09374999999*c*k-0.03125000000*d*k+0.013888888889*b*d
                          -0.28125000000*k*b+
280                     0.08506944444*b*b+0.00173611111*d*d+0.006944444444*c*d;
281       k=fine_mesh_coeff[2];
282       double error2=0.25000000000*k*k-0.09374999999*k*b+0.001736111111*a*a
              +0.00694444444*a*b+
283                     0.01215277778*b*b-0.03125000000*k*a+0.085069444444*c*c
                          +0.01215277778*d*d+
284                     0.04861111111*c*d+0.01388888889*a*c+0.006944444444*a*d
                          +0.04861111111*b*c+
285                     0.01388888889*b*d-0.28125000000*c*k-0.093749999999*d*k;
286       k=fine_mesh_coeff[3];
287       double error3=0.25000000000*k*k-0.03125000000*k*b+0.012152777778*a*a
              +0.00694444444*a*b+
288                     0.00173611111*b*b-0.09374999999*k*a+0.012152777778*c*c
                          +0.08506944444*d*d+
289                     0.04861111111*c*d+0.01388888889*a*c+0.048611111111*a*d
                          +0.00694444444*b*c+
290         0.01388888889*b*d-0.09374999999*c*k-0.281250000000*d*k;
291       std::vector<Point3D> vertices=large_mesh[l]->get_vertices();
292       double diameter=vertices[0].distance(vertices[2]);
293       if(diameter<vertices[1].distance(vertices[3]))
294         diameter=vertices[1].distance(vertices[3]);
295       double hmin=vertices[0].distance(vertices[1]);
296       if(hmin>vertices[0].distance(vertices[3]))
297         hmin=vertices[0].distance(vertices[3]);
298       error_mu_M[l]=hmin*(error0+error1+error2+error3)*((large_mesh[l]->gram_det()))
```

```
        ;
299       mu_M+=error_mu_M[l];
300       free(fine_mesh_coeff);
301     }
302     mu_M=sqrt(mu_M);
303     std::cout<<"Done!␣mu_M="<<mu_M<<"\n\n";
304
305     //---------------------------------//
306     //Compute the error-estimator eta_pi//
307     //---------------------------------//
308     std::cout<<"Starting␣computation␣of␣eta_pi\n";
309     double eta_pi=0.;
310     double* z=(double*)malloc(4*large_mesh.size()*sizeof(double));
311     for (int i=0;i<4*large_mesh.size();++i)
312       z[i]=x_large[i];
313     for (int i=0;i<4*large_mesh.size();++i)
314       eta_pi+=z[i]*c[i];
315     eta_pi=2*eta_pi;
316     double* Bz=B*x_large;
317     double eta_pi_temp=0.;
318     for(int i=0;i<4*large_mesh.size();++i)
319       eta_pi_temp+=x_large[i]*Bz[i];
320     eta_pi=enorm-eta_pi+eta_pi_temp;
321     std::cout<<"Done....␣eta_pi="<<sqrt(eta_pi)<<"\n\n";
322
323     //---------------------------------//
324     //Refine the coarse mesh using mu_pi//
325     //---------------------------------//
326     std::cout<<"Refining␣the␣coarse␣mesh\n";
327     t1=clock();
328     for(int l=0;l<large_mesh.size();++l)
329     {
330       std::vector<double> fine_mesh_coeff=std::vector<double>(4);
331       for(int j=0;j<4;++j)
332         fine_mesh_coeff[j]=x[(large_fine[l])[j]];
333       double b=-0.25*fine_mesh_coeff[0]-0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]+
334           0.25*fine_mesh_coeff[3];
335       double c=-0.25*fine_mesh_coeff[0]+0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]-
336           0.25*fine_mesh_coeff[3];
337       double d=0.25*fine_mesh_coeff[0]-0.25*fine_mesh_coeff[1]+0.25*fine_mesh_coeff
              [2]-
338           0.25*fine_mesh_coeff[3];
339       if(error[l]>=theta*max_error)
340       {
341         if(fabs(tau*b)>sqrt(fabs(c)*fabs(c)+fabs(d)*fabs(d)))
342           large_mesh[l]->mark_red_y();
343         else if(fabs(tau*c)>sqrt(fabs(b)*fabs(b)+fabs(d)*fabs(d)))
344           large_mesh[l]->mark_red_x();
345         else
346           large_mesh[l]->mark_red();
347       }
348     }
349     large_mesh.refine();
350     large_mesh.regulate_diameters(4);
351     t2=clock();
352     std::cout<<"Mesh␣was␣refined.␣Operation␣took␣"<<double((t2-t1)/CLOCKS_PER_SEC)<<
          "␣sec.\n\n";
353
354     //---------------------------------//
355     //Write all the results to a log-file//
356     //---------------------------------//
```

```
357     expfile.open(FILENAME,std::ios::app);
358     expfile.precision(12);
359     expfile<<fine_mesh.size()<<"\t␣␣"<<fabs(enorm)<<"\t␣␣"<<fabs(error2)
360         <<"\t␣␣"<<sqrt(ENORMSQUARE-enorm)
361         <<"\t"<<sqrt(eta_M)<<"\t"<<sqrt(eta_pi)<<"\t"<<mu_M<<"␣\t"<<total_error<<"\n
                ";
362     expfile.close();
363
364     //-------------------------//
365     //Free all allocated memory//
366     //-------------------------//
367     free(b);
368     free(x_large);
369     free(x);
370     free(uH);
371     free(BuH);
372     free(c);
373     free(y);
374     free(z);
375     free(Bz);
376   }
377   std::cout<<"Thank␣you␣for␣using␣FLBEM!\n";
378 }
```

A.3 Triangulierung

A.3.1 geometrics.h

```
1  //************************************************************************//
2  //File:                        geometrics.h                                //
3  //This header file contains the definition of basic geometric classes and  //
4  //functions needed throughout the entire library FLBEM. All definitions are//
5  //contained in the namespace geometrics.                                   //
6  //************************************************************************//
7  //Author: Samuel Ferraz-Leite                                              //
8  //Date: 07.2007                                                            //
9  //Version: 1.0 stable                                                      //
10 //************************************************************************//
11
12 #ifndef BEMLIB_GEOMETRICS_H
13 #define BEMLIB_GEOMETRICS_H
14
15 //lapack.h contains the C-compatible declaration of dgesv
16 extern "C"{
17 #include "lapack.h"
18 }
19
20 namespace geometrics{
21
22 //************************************************************************//
23 // Declaration of classes                                                  //
24 //************************************************************************//
25
26 class Point3D;
27 //Point3D represents a point in the 3-dimensional euklidian space.
28 //standard operators are also implemented for this class.
29
30 typedef Point3D Vector;
31
32 class Edge;
33 //An Edge represents the connecting edge between two Point3D. The
34 //comparison operator is also implemented for this class. An Edge can
35 //tell, if it is the masteredge of another edge. To be master edge
```

```
36 //means to be larger then the edge compaired to in the matter that
37 //is possible during mesh-refinement and with hanging nodes.
38
39 class Matrix;
40 //Matrix represents arbitrary full matrices. The data is stored in the
41 //usual fortran format to provide full compatibility to all lapack routines.
42
43 //************************************************************************//
44 // Definition of classes                                                  //
45 //************************************************************************//
46
47 //---------------//
48 // Class Point3D //
49 //---------------//
50 class Point3D
51 {
52   double x_koord,y_koord,z_koord; //coordinates of the point.
53
54   public:
55     Point3D(){x_koord=0; y_koord=0; z_koord=0;} //Default constructor (0,0,0)
56     Point3D(double x,double y,double z) //Constructs (x,y,z)
57     { x_koord=x; y_koord=y; z_koord=z; }
58     ~Point3D(){}//The destructor has actually no work to do
59
60     double x() const {return x_koord;} //These are the default get functions
61     double y() const {return y_koord;} //to read the koordinates.
62     double z() const {return z_koord;}
63
64     void set_x(double x){x_koord=x;} //These are the default set
65     void set_y(double y){y_koord=y;} //functions to set the koordinates
66     void set_z(double z){z_koord=z;}
67
68     double distance(const Point3D& p) const;//calculates the distance between
69                                             //this and a given Point3D p
70
71     Point3D operator* (double a) //scaling coords with a scalar a
72     {return Point3D(x_koord*a,y_koord*a,z_koord*a);}
73     Point3D operator/ (double a) //scaling coords with a scalar a
74     {return Point3D(x_koord/a,y_koord/a,z_koord/a);}
75 };
76
77 //-----------------------//
78 // Operators for Point3D //
79 //-----------------------//
80
81 //The operator < is provided to enable the use of maps and other containers who
82 //need ordering routines.
83 inline bool operator<(const Point3D& p, const Point3D& q)
84 {
85   if(p.x()<q.x())
86     return true;
87   else if((p.x() == q.x()) && (p.y() < q.y()))
88     return true;
89   else if((p.x() == q.x()) && (p.y() == q.y()) && (p.z()<q.z()))
90     return true;
91   else
92     return false;
93 }
94
95 inline bool operator==(const Point3D& p, const Point3D& q)
96 {return ((p.x()==q.x()) && (p.y()==q.y()) && (p.z()==q.z()));}
97
98 //+ and - implement the componentwise addition and subtraction
```

```
99  inline Point3D operator+ (const Point3D& p,const Point3D& q)
100 {return Point3D(p.x()+q.x(),p.y()+q.y(),p.z()+q.z());}
101
102 inline Point3D operator- (const Point3D& p,const Point3D& q)
103 {return Point3D(p.x()-q.x(),p.y()-q.y(),p.z()-q.z());}
104
105 //------------//
106 // Class Edge //
107 //------------//
108 class Edge
109 {
110   protected:
111     Point3D p;
112     Point3D q;
113
114   public:
115     Edge() : p(),q(){}//Standard constructor [(0,0,0),(0,0,0)]
116     Edge(Point3D point1,Point3D point2){p=point1;q=point2;}//makes edge [P1,P2]
117     Edge(double x1,double y1,double z1, //construct the edge
118          double x2,double y2,double z2) : p(x1,y1,z1), q(x2,y2,z2)
119     {}//[(x1,y1,z1),(x2,y2,z2)]
120     ~Edge(){p.~Point3D();q.~Point3D();}//destructor
121     Point3D p1() const{return p;}//get p
122     Point3D p2() const{return q;}//get q
123     void set_p1(Point3D point){p=point;}//set p
124     void set_p2(Point3D point){q=point;}//set q
125     bool is_masteredge_of(Edge) const;//checks if this is larger than the
126                                       //parameter edge
127 };
128
129 //--------------------//
130 // Operators for Edge //
131 //--------------------//
132
133 //Two Edge Objects are considered to be identical, if they have the same
134 //endpoints independent of the orientation. This means [a,b]==[b,a].
135 inline bool operator==(const Edge& e, const Edge& f)
136 {return ((f.p1()==e.p1() && f.p2()==e.p2())||(f.p1()==e.p2() && f.p2()==e.p1()));}
137
138 inline bool operator<(const Edge& e1,const Edge& e2)//Operator for Edge class.
139 {
140   if(e1==e2)
141     return false;
142   else if(e1.p1()<e2.p1())
143     return true;
144   else if((e1.p1()==e2.p1()) && (e1.p2()<e2.p2()))
145     return true;
146   else
147     return false;
148 }
149
150 //--------------//
151 // Class Matrix //
152 //--------------//
153 class Matrix
154   {
155   int rdim,cdim;
156   public:
157   double* data;
158     Matrix();
159     Matrix(int n);
160     Matrix(int m,int n);
161     ~Matrix();
```

```
162     void set(int i,int j,double value);
163     double get(int i,int j) const;
164     int get_rowdim() const;
165     int get_coldim() const;
166 };
167 
168 //*************************************************************************//
169 // Usefull functions for geometric manipulations                           //
170 //*************************************************************************//
171 double norm(const Vector&);
172 double squarenorm(const Vector&);
173 double scalarproduct(const Vector&,const Vector&);
174 Vector vectorproduct(const Vector&,const Vector&);
175 
176 //Normalize converts a Vector object to become the corresponding
177 //vector with Euclidian norm 1
178 void normalize(Vector&);
179 
180 //MVM denotes the matrix-vector-multiplikation. It is implemented
181 //for 3-dimensional manipulations and for arbitrary dimensions.
182 Vector MVM(const Matrix&,const Vector&);
183 double* MVM(const Matrix& A, double* v);
184 
185 //Note! The Gauss-Algorithm to solve a linear equation alteres the
186 //data-field of the Matrix A object.
187 double* solveGauss(Matrix& A, double* b);
188 
189 }
190 
191 #endif
```

A.3.2 geometrics.cpp

```
1  //*************************************************************************//
2  //File:                        geometrics.cpp                              //
3  //This file implements all functions defined in geometrics.cpp. The       //
4  //purpose of all routines found in this file are explained in the          //
5  //corresponding header.                                                    //
6  //*************************************************************************//
7  //Author: Samuel Ferraz-Leite                                              //
8  //Date: 07.2007                                                            //
9  //Version: 1.0 stable                                                      //
10 //*************************************************************************//
11 
12 //Includes of the needed standard-libraries
13 #include<cmath>
14 #include<vector>
15 #include<list>
16 #include<iostream>
17 
18 //Includes of the needed FLBEM-headers
19 #include "geometrics.h"
20 
21 namespace geometrics{
22 //*************************************************************************//
23 // Implementation of functions in Matrix                                   //
24 //*************************************************************************//
25 Matrix::Matrix()
26 {
27   data=NULL;
28   rdim=0;
29   cdim=0;
30 }
31 
```

```
32 Matrix::Matrix(int n)
33 {
34   data=(double*)malloc((n*n)*sizeof(double));
35   for(int i=0;i<n*n;++i)
36     data[i]=0.;
37   cdim=n;
38   rdim=n;
39 }
40
41 Matrix::Matrix(int m,int n)
42 {
43   data=(double*)malloc((m*n)*sizeof(double));
44   for(int i=0;i<m*n;++i)
45     data[i]=0.;
46   rdim=m;
47   cdim=n;
48 }
49
50 Matrix::~Matrix()
51 {
52   free(data);
53 }
54
55 void Matrix::set(int i,int j,double value)
56 {
57   data[(j-1)*rdim+(i-1)]=value;
58 }
59
60 double Matrix::get(int i,int j) const
61 {
62   return data[(j-1)*rdim+(i-1)];
63 }
64
65 int Matrix::get_rowdim() const
66 {return rdim;}
67
68 int Matrix::get_coldim() const
69 {return cdim;}
70
71
72
73 //************************************************************************//
74 // Implementation of functions in Point3D                                 //
75 //************************************************************************//
76 double Point3D::distance(const Point3D& p) const
77 {
78   return
79       (sqrt((p.x()-x_koord)*(p.x()-x_koord)+
80       (p.y()-y_koord)*(p.y()-y_koord)+
81       (p.z()-z_koord)*(p.z()-z_koord)));
82 }
83
84 //************************************************************************//
85 // Implementation of functions in Edge                                    //
86 //************************************************************************//
87 bool Edge::is_masteredge_of(Edge edge_to_check) const
88 {
89   if ((*this)==edge_to_check)
90     return true;
91   if (Edge(p,(p+q)/2.)==edge_to_check)
92     return true;
93   if (Edge(q,(p+q)/2.)==edge_to_check)
94     return true;
```

```
95    if(Edge(p,(((p+q)/2.)+p)/2.)==edge_to_check)
96      return true;
97    if(Edge((((p+q)/2.)+p)/2.,(p+q)/2.)==edge_to_check)
98      return true;
99    if(Edge((p+q)/2.,(((p+q)/2.)+q)/2.)==edge_to_check)
100     return true;
101   if(Edge((((p+q)/2.)+q)/2.,q)==edge_to_check)
102     return true;
103   return false;
104 }
105
106
107 double norm(const Vector& a)
108 {
109   return sqrt(a.x()*a.x()+a.y()*a.y()+a.z()*a.z());
110 }
111
112 double squarenorm(const Vector& a)
113 {
114   return (a.x()*a.x()+a.y()*a.y()+a.z()*a.z());
115 }
116
117 double scalarproduct(const Vector& a,const Vector& b)
118 {
119   return (a.x()*b.x()+a.y()*b.y()+a.z()*b.z());
120 }
121
122 Vector vectorproduct(const Vector& a,const Vector& b)
123 {
124   return Vector(a.y()*b.z()-b.y()*a.z(),a.x()*b.z()-b.x()*a.z(),a.x()*b.y()-b.x()*a.
        y());
125 }
126
127
128 //************************************************************************//
129 // Implementation of other routines                                       //
130 //************************************************************************//
131 void normalize(Vector& a)
132 {
133   double n=norm(a);
134   a.set_x(a.x()/n);
135   a.set_y(a.y()/n);
136   a.set_z(a.z()/n);
137 }
138
139 Vector MVM(const Matrix& A,const Vector& x)
140 {
141   Vector result=Vector(0,0,0);
142   result.set_x(A.get(1,1)*x.x()+A.get(1,2)*x.y()+A.get(1,3)*x.z());
143   result.set_y(A.get(2,1)*x.x()+A.get(2,2)*x.y()+A.get(2,3)*x.z());
144   result.set_z(A.get(3,1)*x.x()+A.get(3,2)*x.y()+A.get(3,3)*x.z());
145   return result;
146 }
147
148 //This is the generalized Version of the matrix-vector product for vectors
149 //of arbitraty length. The length of the vector is assumed to be correct with
150 //respect to the col-dimension of the matrix.
151 double* MVM(const Matrix& A, double* v)
152 {
153   double* result=(double*)malloc(A.get_rowdim()*sizeof(double));
154   for(int i=0;i<A.get_rowdim();++i)
155   {
156     result[i]=0;
```

```
157       for(int j=0;j<A.get_coldim();++j)
158         result[i]+=A.get(i+1,j+1)*v[j];
159     }
160     return result;
161   }
162
163   //This function basically calls dgesv of lapack to compute the Gauss-Algorithm.
164   //All dimensions are assumed to be correct and the Matrix A is assumed
165   //to be regular and square.
166   double* solveGauss(Matrix& A,double* b)
167   {
168     int n=A.get_rowdim();
169     double* btilde=(double*)malloc(n*sizeof(double));
170     for(int i=0;i<n;++i)
171       btilde[i]=b[i];
172     for(int i=0;i<n;++i)
173     {
174       double factor=1./fabs(A.get(i+1,i+1));
175       btilde[i]*=factor;
176       for (int j=0;j<n;++j)
177         A.set(i+1,j+1,A.get(i+1,j+1)*factor);
178     }
179     int d=1;
180     int nA=n;
181     int nB=n;
182     double* a=A.data;
183     int* Q=(int*)malloc(n*sizeof(int));
184     for(int i=0;i<n;++i)
185       Q[i]=i;
186     int info=0;
187     dgesv_(&n,&d,a,&nA,Q,btilde,&nB,&info);
188     if(info!=0)
189       std::cout<<"Warning!␣Something␣happened␣while␣solving␣with␣Gauss!\n";
190     return btilde;
191   }
192
193   }
```

A.3.3 triangulation.h

```
1  //************************************************************************//
2  //File:                    triangulation.h                                  //
3  //This is the header file for the triangulation module of FLBEM. The file  //
4  //containes the deklaration and definition of objects and methods used to  //
5  //represent an adaptive triangulation with triangles or rectangles of a    //
6  //2-dimensional manifold in a 3-dimensional space, as needed in            //
7  //BEM-calculations.                                                        //
8  //************************************************************************//
9  //Author: Samuel Ferraz-Leite                                              //
10 //Date: 07.2007                                                           //
11 //Version 0.8 not all features implemented for Triangles                  //
12 //************************************************************************//
13 #ifndef BEMLIB_TRIANGULATION_H
14 #define BEMLIB_TRIANGULATION_H
15
16 //inlcudes of other FLBEM-modules
17 #include "geometrics.h"
18
19 //Includes of the standard-libraries
20 #include<vector>
21 #include<list>
22
23 //For clarity and better modul-splitting the whole library is
24 //contained in the namespace tr3D.
```

```
25 namespace triangulation{
26
27 using namespace geometrics;
28
29 //Some enum-types needed to transport information about the Shape-type and the
30 //ansatz-functions used for the BEM
31 enum border_type {DIRICHLET, NEUMANN};
32
33 enum shape_type {RECTANGLE, TRIANGLE};
34
35 enum function_type {CONSTANT, LINEAR, LINCONST, CONSTLIN};
36
37 //************************************************************************//
38 // Declaration of classes                                                 //
39 //************************************************************************//
40
41 class ShapeEdge;
42 //A ShapeEdge is a special form of Edge which represents the Edges
43 //of a triangle or a rectangle. ShapeEdges have all necessary
44 //routines and variables for mesh refinement. It knows who its
45 //neighbour Shapes (triangles or rectangles) are an can tell if
46 //another edge corresponds to it. Corresponding to a ShapeEdge means,
47 //that either the ShapeEdge is a master-edge or the Edge is a master-
48 //edge.
49
50 class Shape;
51 //The Shape-class is the abstract class for triangles and rectangles.
52 //A shape is defined by its ShapeEdges. The Shape can be marked red,
53 //it can split itself, neighbours can be added or removed, it can
54 //process the information properly, if a neighbours ShapeEdge changes
55 //and it can check if a Shape is still a neighbour.
56
57 class Triangle;
58 //The Triangle class is a concrete implementation of the abstract
59 //Shape class.
60
61 class Rectangle;
62 //The Rectangle class is a concrete implementation of the abstract
63 //Shape class.
64
65 class Triangulation3D;
66 //This class is a container of pointer to Shapes. It is a descendant
67 //of the std::vector class providing a constructor, which allows to
68 //create the first triangulation specified by an initial Shape-Vector. The
69 //Shapes contained by the Triangulation3D must be marked red from
70 //outside. After having marked red all relevant triangles the method
71 //refine() can be called, which causes the mesh to refine itself.
72
73 //************************************************************************//
74 //Definition of the classes.                                              //
75 //************************************************************************//
76
77 //-----------------//
78 // Calss ShapeEdge //
79 //-----------------//
80 class ShapeEdge : public Edge
81 {
82   bool marked_to_split; //Set true if this edge is to be split.
83   std::list<Shape*> neighbours;//All though an edge may only have 2 neighbours
84                                //it may happend, that temporarily up to 4
85                                //neigbours exist.
86   std::list<Point3D> hanging_nodes;//the hanging nodes on this edge.
87   public:
```

```
88      ShapeEdge(); //Standard constructor [(0,0,0),(0,0,0)]
89      ShapeEdge(Point3D,Point3D); //constructs the edge [P,Q].
90      ShapeEdge(double,double,double, //construct the edge
91                double,double,double); //[(x1,y1,z1),(x2,y2,z2)]
92      ~ShapeEdge();//standard destructor
93      void add_neighbour(Shape*);
94      void remove_neighbour(Shape* neighbour)
95        {neighbours.remove(neighbour);}
96      void add_hanging(Point3D);//add a hanging node, if it really is on the edge.
97      void remove_hanging(Point3D point)
98        {hanging_nodes.remove(point);}
99      std::list<Point3D> get_hanging_nodes() const;
100     bool is_hanging() const
101       {return (hanging_nodes.size()!=0);}//check if edge has hanging nodes
102     bool mark_to_split();//mark this edge to be split - return true, if its
103                          //status changed (i.e. it wasnt marked to be split
104                          //before.)
105     void unmark_to_split(){marked_to_split=false;}
106     bool is_to_split() const{return marked_to_split;}
107     bool corresponds_to(const Edge& edge_to_check) const
108       {return(is_masteredge_of(edge_to_check) ||
109               edge_to_check.is_masteredge_of((*this)));}
110     void send();//contacts all the neighbours so that they may change their own
111                 //status according to the necessity of limiting the number of
                        hanging
112                 //nodes
113     int number_of_neighbours() const
114       {return neighbours.size();}
115 };
116
117 //-------------//
118 // Class Shape //
119 //-------------//
120 class Shape
121 {
122   public:
123     virtual std::vector<ShapeEdge*> get_edges()=0;
124     virtual std::vector<Point3D> get_vertices() const=0;
125     virtual std::vector<Shape*> get_neighbours()=0;
126     virtual void add_neighbour(Shape*)=0;//adds a neighbour, if it really is one
127     virtual void remove_neighbour(Shape*)=0;//removes the neighbour
128     virtual void mark_red(bool contact_neighbours=true)=0;//marks all edges to be
129                                                           //split and lets the
130                                                           //edges send if
131                                                           //contact_neighbours==true
132     virtual void mark_red_x(bool contact_neighbours=true)=0;//Anisotropic
133     virtual void mark_red_y(bool contact_neighbours=true)=0;//refinement for
134                                                             //rectangles only
135
136     virtual bool is_to_split() const=0;//tells if shape is marked to split
137     virtual std::vector<Shape*> split(bool consider_neighbourhood=true)=0;
138     virtual void allow_hanging()=0;//sets hanging_allowed to true
139     virtual void listen_edge(ShapeEdge*)=0;//makes necessary changes on the Shape
140                                            //acording to the status of the ShapeEdge
141     virtual void check_neighbourhood(Shape*)=0;//checks if Shape is still a
          neighbour
142     virtual void display_information() const=0;//Displays some basic info
143                                                //like coords of the corners
144     virtual border_type border_information() const=0;//returns the border_type of a
          Shape
145     virtual shape_type what_are_you() const=0;//returns the Shape-type
146     virtual double gram_det() const=0;
147 };
```

```
148
149
150 //-----------------//
151 // Class Triangle //
152 //-----------------//
153 //Note: A triangle allways assumes, that its neighbours are also Triangles!
154 //Note: The constructors without bool variable argument set
155 //hanging_allowed=false.
156 class Triangle : public virtual Shape
157 {
158   ShapeEdge ab;
159   ShapeEdge bc;
160   ShapeEdge ca;
161   border_type my_border_type;
162   bool hanging_allowed;
163   bool marked_red;
164   std::list<Shape*> neighbours;
165   ShapeEdge* find_neighbour_edge(Shape*);
166   ShapeEdge* longest_edge();
167   public:
168     Triangle() : ab(),bc(),ca(),neighbours()
169       {my_border_type=DIRICHLET;hanging_allowed=false;marked_red=false;}
170     Triangle(bool hanging) : ab(),bc(),ca(),neighbours()
171       {my_border_type=DIRICHLET;hanging_allowed=hanging;marked_red=false;}
172     Triangle(Point3D,Point3D,Point3D);
173     Triangle(Point3D,Point3D,Point3D,bool);
174     Triangle(Point3D,Point3D,Point3D,border_type);
175     Triangle(Point3D,Point3D,Point3D,bool,border_type);
176     Triangle(const Triangle&);//Does not copy the neighbourhood-pointers!
177     ~Triangle();
178     Triangle& operator=(const Triangle&);//Des not copy the neighbourhood-pointers!
179                                          //additionally it deletes the old pointers!
180     std::vector<Point3D> get_vertices() const;
181     std::vector<ShapeEdge*> get_edges();
182     std::vector<Shape*> get_neighbours();
183     void add_neighbour(Shape*);
184     void remove_neighbour(Shape* neighbour)
185       {
186         neighbours.remove(neighbour);
187         ab.remove_neighbour(neighbour);
188         bc.remove_neighbour(neighbour);
189         ca.remove_neighbour(neighbour);
190       }
191     void mark_red(bool contact_neighbours=true);
192     void mark_red_x(bool contact_neighbours=true);
193     void mark_red_y(bool contact_neighbours=true);
194     bool is_to_split() const
195     {return(((Triangle*)(this))->longest_edge()->is_to_split());}//Note this
196                       //conversion is done due to perserving the const
197                       //attribute of this function but keeping up the
198                       //philosophy, that getting pointers to edges should
199                       //not be const, since the user might want to get them
200                       //to change the objects properties.
201     std::vector<Shape*> split(bool consider_neighbourhood=true);
202     void allow_hanging()
203       {hanging_allowed=true;}
204     void listen_edge(ShapeEdge*);
205     void check_neighbourhood(Shape*);
206     void display_information() const;
207     border_type border_information() const{return my_border_type;}
208     bool is_hanging_allowed() const {return hanging_allowed;};
209     double gram_det() const;
210     shape_type what_are_you() const {return TRIANGLE;};
```

```
211 };
212
213 //------------------//
214 // Class Rectangle //
215 //------------------//
216 //Note: A Rectangle allways assumes, that its neighbours are also
217 //Rectangles!
218 //Note: The constructors without bool variable argument set
219 //hanging_allowed=true.
220 class Rectangle : public virtual Shape
221 {
222   ShapeEdge ab;//a field of edges
223   ShapeEdge bc;
224   ShapeEdge cd;
225   ShapeEdge da;
226   border_type my_border_type;
227   bool hanging_allowed;
228   bool marked_red;
229   std::list<Shape*> neighbours;
230   ShapeEdge* find_neighbour_edge(Shape*);
231   public:
232     Rectangle();
233     Rectangle(bool);
234     Rectangle(Point3D,Point3D,Point3D,Point3D);
235     Rectangle(Point3D,Point3D,Point3D,Point3D,bool);
236     Rectangle(Point3D,Point3D,Point3D,Point3D,border_type);
237     Rectangle(Point3D,Point3D,Point3D,Point3D,bool,border_type);
238     Rectangle(const Rectangle&);//Does not copy the neighbour-pointers!
239     ~Rectangle();
240     std::vector<Point3D> get_vertices() const;
241     std::vector<ShapeEdge*> get_edges();
242     std::vector<Shape*> get_neighbours();
243     void add_neighbour(Shape*);
244     void remove_neighbour(Shape*);
245     void mark_red(bool contact_neighbours=true);
246     void mark_red_x(bool contact_neighbours=true);
247     void mark_red_y(bool contact_neighbours=true);
248     bool is_to_split() const;
249     bool is_hanging_allowed() const{return hanging_allowed;};
250     std::vector<Shape*> split(bool consider_neighbourhood=true);
251     void allow_hanging();
252     void listen_edge(ShapeEdge*);
253     void check_neighbourhood(Shape*);
254     void display_information() const;
255     border_type border_information() const;
256     double gram_det() const;
257     shape_type what_are_you() const {return RECTANGLE;};
258 };
259
260 //-----------------------//
261 // Class Triangulation3D //
262 //-----------------------//
263 class Triangulation3D : public std::vector<Shape*>
264 {
265   bool consider_neighbourhood;
266   public:
267     void* data;
268     Triangulation3D();//create an empty triangulation
269     Triangulation3D(std::vector<Shape*>);//Triangulation containing the Elements of
270                                          //the list
271     Triangulation3D(std::vector<Shape*>,bool set_neighbours);
272     Triangulation3D(std::string);//create a triangulation from a file
273     Triangulation3D(const Triangulation3D&);//Will make a deep copy of the mesh.
```

```
274     ~Triangulation3D();//destroy all the Triangles and the Triangiulation itself
275     Triangulation3D& operator=(const Triangulation3D&);
276     bool is_considering_neighbourhood() const;
277     void regulate_diameters(double k);//Makes sure, that no neighbouring elements
278                                       //differ in diameter by more than a factor k.
279                                       //So far only implemented for RECTANGULAR
280                                       //meshes!
281     void refine();//splits all Shapes, that are marked to be split.
282     void refine_uniformily();//markes and splits every Shape of the triangulation
283                              //red. This routine should be faster, than trying
284                              //to mark every shape red first and then
285                              //refining the mesh with refine().
286 };
287
288 }
289
290 #endif
```

A.3.4 triangulation.cpp

```
1  //************************************************************************//
2  //File:                    triangulation.cpp                              //
3  //This file implements the classes and functions specified in triangulation.h //
4  //************************************************************************//
5  //Author: Samuel Ferraz-Leite                                             //
6  //Date: 07.2007                                                           //
7  //Version 0.8 not all features implemented for Triangles                  //
8  //************************************************************************//
9
10 //Includes of the standard-libraries
11 #include<cmath>
12 #include<vector>
13 #include<list>
14 #include<iostream>
15
16 //Includes of FLBEM-modules
17 #include "triangulation.h"
18 #include "geometrics.h"
19
20 namespace triangulation{
21
22 //************************************************************************//
23 // Implementation of functions in ShapeEdge                               //
24 //************************************************************************//
25 ShapeEdge::ShapeEdge()
26 {
27   set_p1(Point3D());
28   set_p2(Point3D());
29   marked_to_split=false;
30   hanging_nodes=std::list<Point3D>();
31   neighbours=std::list<Shape*>();
32 }
33
34 ShapeEdge::ShapeEdge(Point3D point1,Point3D point2)
35 {
36   set_p1(point1);
37   set_p2(point2);
38   marked_to_split=false;
39   hanging_nodes=std::list<Point3D>();
40   neighbours=std::list<Shape*>();
41 }
42
43 ShapeEdge::ShapeEdge(double x1,double y1,double z1,
44                      double x2,double y2,double z2)
```

```
45 {
46   set_p1(Point3D(x1,y1,z1));
47   set_p2(Point3D(x2,y2,z2));
48   marked_to_split=false;
49   hanging_nodes=std::list<Point3D>();
50   neighbours=std::list<Shape*>();
51 }
52
53 ShapeEdge::~ShapeEdge()
54 {
55   neighbours.~list<Shape*>();
56   hanging_nodes.~list<Point3D>();
57 }
58
59 void ShapeEdge::add_neighbour(Shape* neighbour)
60 {
61   if(find(neighbours.begin(),neighbours.end(),neighbour)==neighbours.end())
62     neighbours.push_back(neighbour);
63 }
64
65 void ShapeEdge::add_hanging(Point3D point)
66 {
67  if((find(hanging_nodes.begin(),hanging_nodes.end(),point)==hanging_nodes.end()) &&
68       ((point==(p+q)/2.) || (point==(p+(p+q)/2.)/2.) ||
69       (point==(q+(p+q)/2.)/2.)))
70     hanging_nodes.push_back(point);
71 }
72
73 std::list<Point3D> ShapeEdge::get_hanging_nodes() const
74 {return hanging_nodes;}
75
76 bool ShapeEdge::mark_to_split()
77 {
78   bool return_var=!(marked_to_split);
79   marked_to_split=true;
80   return return_var;
81 }
82
83 void ShapeEdge::send()
84 {
85   for(std::list<Shape*>::iterator i=neighbours.begin();i!=neighbours.end();++i)
86   {
87     (*i)->listen_edge(this);
88   }
89 }
90
91
92 //************************************************************************//
93 // Implementation of functions in Triangle                                //
94 //************************************************************************//
95 ShapeEdge* Triangle::find_neighbour_edge(Shape* neighbour)
96 {
97   //get the edges of the neighbour
98   std::vector<ShapeEdge*> neighbour_edges=neighbour->get_edges();
99
100  if(ab.corresponds_to(*neighbour_edges[0]) || ab.corresponds_to(*neighbour_edges
       [1])
101     || ab.corresponds_to(*neighbour_edges[2]))
102    return &ab;
103  if(bc.corresponds_to(*neighbour_edges[0]) || bc.corresponds_to(*neighbour_edges
       [1])
104     || bc.corresponds_to(*neighbour_edges[2]))
105    return &bc;
```

```
106   if(ca.corresponds_to(*neighbour_edges[0]) || ca.corresponds_to(*neighbour_edges
          [1])
107      || ca.corresponds_to(*neighbour_edges[2]))
108     return &ca;
109   return 0;
110 }
111
112 ShapeEdge* Triangle::longest_edge()
113 {
114   ShapeEdge* longest=&ab;
115   if(longest->p1().distance(longest->p2())<bc.p1().distance(bc.p2()))
116     longest=&bc;
117   if(longest->p1().distance(longest->p2())<ca.p1().distance(ca.p2()))
118     longest=&ca;
119   return longest;
120 }
121
122
123 Triangle::Triangle(Point3D a,Point3D b,Point3D c)
124 {
125   ab=ShapeEdge(a,b);
126   bc=ShapeEdge(b,c);
127   ca=ShapeEdge(c,a);
128   hanging_allowed=false;
129   marked_red=false;
130   neighbours=std::list<Shape*>();
131   my_border_type=DIRICHLET;
132 }
133
134 Triangle::Triangle(Point3D a,Point3D b,Point3D c,bool hanging)
135 {
136   ab=ShapeEdge(a,b);
137   bc=ShapeEdge(b,c);
138   ca=ShapeEdge(c,a);
139   hanging_allowed=hanging;
140   marked_red=false;
141   neighbours=std::list<Shape*>();
142   my_border_type=DIRICHLET;
143 }
144
145 Triangle::Triangle(Point3D a,Point3D b,Point3D c,border_type neumann_or_dirichlet)
146 {
147   ab=ShapeEdge(a,b);
148   bc=ShapeEdge(b,c);
149   ca=ShapeEdge(c,a);
150   hanging_allowed=false;
151   marked_red=false;
152   neighbours=std::list<Shape*>();
153   my_border_type=neumann_or_dirichlet;
154 }
155
156 Triangle::Triangle(Point3D a,Point3D b,Point3D c,bool hanging,border_type
      neumann_or_dirichlet)
157 {
158   ab=ShapeEdge(a,b);
159   bc=ShapeEdge(b,c);
160   ca=ShapeEdge(c,a);
161   hanging_allowed=hanging;
162   marked_red=false;
163   neighbours=std::list<Shape*>();
164   my_border_type=neumann_or_dirichlet;
165 }
166
```

```
167 Triangle::Triangle(const Triangle& T)
168 {
169   std::vector<Point3D> points=T.get_vertices();
170   ab=ShapeEdge(points[0],points[1]);
171   bc=ShapeEdge(points[1],points[2]);
172   ca=ShapeEdge(points[2],points[0]);
173   hanging_allowed=T.is_hanging_allowed();
174   marked_red=T.is_to_split();
175   neighbours=std::list<Shape*>();
176   my_border_type=T.border_information();
177 }
178
179 Triangle::~Triangle()
180 {
181   ab.~ShapeEdge();
182   bc.~ShapeEdge();
183   ca.~ShapeEdge();
184   neighbours.~list<Shape*>();
185 }
186
187 Triangle& Triangle::operator=(const Triangle& T)
188 {
189   if(this!=&T)
190   {
191     std::vector<Point3D> points=T.get_vertices();
192     ab=ShapeEdge(points[0],points[1]);
193     bc=ShapeEdge(points[1],points[2]);
194     ca=ShapeEdge(points[2],points[0]);
195     hanging_allowed=T.is_hanging_allowed();
196     marked_red=T.is_to_split();
197     neighbours=std::list<Shape*>();
198     my_border_type=T.border_information();
199   }
200 }
201
202 std::vector<ShapeEdge*> Triangle::get_edges()
203 {
204   std::vector<ShapeEdge*> edges=std::vector<ShapeEdge*>();
205   edges.push_back(&ab);
206   edges.push_back(&bc);
207   edges.push_back(&ca);
208   return edges;
209 }
210
211 std::vector<Shape*> Triangle::get_neighbours()
212 {
213   std::vector<Shape*> n=std::vector<Shape*>();
214   for(std::list<Shape*>::iterator i=neighbours.begin();i!=neighbours.end();++i)
215     n.push_back(*i);
216   return n;
217 }
218
219 void Triangle::add_neighbour(Shape* neighbour)
220 {
221   ShapeEdge* edge=find_neighbour_edge(neighbour);
222   if(edge!=0)
223   {
224     edge->add_neighbour(neighbour);
225     //add the neighbour to the list, if its not yet listed
226     if(find(neighbours.begin(),neighbours.end(),neighbour)==neighbours.end())
227     {
228       neighbours.push_back(neighbour);
229     }
```

```
230   }
231 }
232
233 void Triangle::mark_red(bool contact_neighbours)
234 {
235   if(ab.mark_to_split() && contact_neighbours)
236     ab.send();
237   if(bc.mark_to_split() && contact_neighbours)
238     bc.send();
239   if(ca.mark_to_split() && contact_neighbours)
240     ca.send();
241   marked_red=true;
242 }
243
244 void Triangle::mark_red_x(bool contact_neighbours)
245 {
246   if(ab.mark_to_split() && contact_neighbours)
247     ab.send();
248   if(bc.mark_to_split() && contact_neighbours)
249     bc.send();
250   if(ca.mark_to_split() && contact_neighbours)
251     ca.send();
252   marked_red=true;
253 }
254
255 void Triangle::mark_red_y(bool contact_neighbours)
256 {
257   if(ab.mark_to_split() && contact_neighbours)
258     ab.send();
259   if(bc.mark_to_split() && contact_neighbours)
260     bc.send();
261   if(ca.mark_to_split() && contact_neighbours)
262     ca.send();
263   marked_red=true;
264 }
265
266
267 std::vector<Shape*> Triangle::split(bool consider_neighbourhood)
268 {
269   std::vector<Shape*> new_triangles=std::vector<Shape*>();//return variable
270   if(marked_red)
271   {
272     marked_red=false;//the modified triangle should not be split
273     //create the new triangles
274     Triangle* tr1=new Triangle(ab.p1(),(ab.p1()+ab.p2())/2.,
275                                (ca.p1()+ca.p2())/2.,hanging_allowed);
276     Triangle* tr2=new Triangle((ab.p1()+ab.p2())/2.,ab.p2(),
277                                (bc.p1()+bc.p2())/2.,hanging_allowed);
278     Triangle* tr3=new Triangle(ca.p1()+ca.p2())/2.,(bc.p1()+bc.p2())/2.,
279                                ca.p1(),hanging_allowed);
280     if(hanging_allowed)//if there might be hanging nodes
281     {//add the hanging nodes of the first edge
282       std::list<Point3D> hanging_nodes=ab.get_hanging_nodes();
283       std::vector<ShapeEdge*> edges1=tr1->get_edges();
284       std::vector<ShapeEdge*> edges2=tr2->get_edges();
285       std::list<Point3D>::const_iterator i;
286       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
287       {
288         edges1[0]->add_hanging(*i);
289         edges2[0]->add_hanging(*i);
290       }
291       //add the hanging nodes of the second edge
292       hanging_nodes=bc.get_hanging_nodes();
```

```
293       edges1=tr2->get_edges();
294       edges2=tr3->get_edges();
295       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
296       {
297         edges1[1]->add_hanging(*i);
298         edges2[1]->add_hanging(*i);
299       }
300       //add the hanging nodes of the third edge
301       hanging_nodes=ca.get_hanging_nodes();
302       edges1=tr3->get_edges();
303       edges2=tr1->get_edges();
304       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
305       {
306         edges1[2]->add_hanging(*i);
307         edges2[2]->add_hanging(*i);
308       }
309     }
310     //correct my own edges
311     ab=ShapeEdge((ab.p1()+ab.p2())/2.,(bc.p1()+bc.p2())/2.);
312     bc=ShapeEdge((bc.p1()+bc.p2())/2.,(ca.p1()+ca.p2())/2.);
313     ca=ShapeEdge((ca.p1()+ca.p2())/2.,ab.p1());
314     //set neighbourhood-relationships
315     std::list<Shape*> old_neighbours=neighbours;
316     for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
317                                           i!=old_neighbours.end();++i)
318     {
319       tr1->add_neighbour(*i);//Add my neighbours to the new triangles
320       tr2->add_neighbour(*i);
321       tr3->add_neighbour(*i);
322       (*i)->remove_neighbour(this);//remove myself from my old neighbours
323       (*i)->add_neighbour(tr1);//add the new triangles as neighbours to the
324       (*i)->add_neighbour(tr2);
325       (*i)->add_neighbour(tr3);
326       remove_neighbour(*i);//remove the neighbour from my list
327     }
328     ca.add_neighbour(tr1);//add the new triangles to my neighbourlist
329     neighbours.push_back(tr1);//and the list of the corresponding edge
330     ab.add_neighbour(tr2);
331     neighbours.push_back(tr2);
332     bc.add_neighbour(tr3);
333     neighbours.push_back(tr3);
334     tr1->add_neighbour(this);//add myself as neighbour to the new
335     tr2->add_neighbour(this);//triangles
336     tr3->add_neighbour(this);
337     //set the return variable
338     new_triangles.push_back(tr1);
339     new_triangles.push_back(tr2);
340     new_triangles.push_back(tr3);
341   }
342   else
343   {
344     //if ab is the longest edge
345     if(longest_edge()==&ab)
346     {
347       //create the new triangle
348       Triangle* tr=new Triangle((ab.p1()+ab.p2())/2.,bc.p1(),bc.p2(),
349                                                       hanging_allowed);
350       //copy information of the edge bc
351       tr->bc=bc;
352       //set my bc correctly
353       bc=ShapeEdge((ab.p1()+ab.p2())/2.,ca.p1());
354       //if hanging nodes are allowed transfer hanging nodes correctly
355       if(hanging_allowed)
```

```
356       {
357         std::list<Point3D> hanging_nodes=ab.get_hanging_nodes();
358         for(std::list<Point3D>::iterator i=hanging_nodes.begin();
359                                          i!=hanging_nodes.end();++i)
360         {
361           if((*i)==(tr->ab.p1()+tr->ab.p2())/2.)
362           {
363             tr->ab.add_hanging(*i);
364             ab.remove_hanging(*i);
365           }
366           if((*i)==((ab.p1()+ab.p2())/2.))
367             ab.remove_hanging(*i);
368         }
369       }
370       //set point b of my ab correctly and unmark for splitting
371       ab.set_p2((ab.p1()+ab.p2())/2.);
372       ab.unmark_to_split();
373       //now update all neighbourhood relationships
374       std::list<Shape*> old_neighbours=neighbours;
375       for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
376                                            i!=old_neighbours.end();++i)
377       {
378         (*i)->check_neighbourhood(this);
379         check_neighbourhood(*i);
380         (*i)->add_neighbour(tr);
381         tr->add_neighbour(*i);
382       }
383       add_neighbour(tr);
384       tr->add_neighbour(this);
385       //if the subtriangles need further to be split call split for the
386       //corresponding one --> note: it cant happen, that both must be
387       //split.
388       if(ca.is_to_split())
389       {
390         new_triangles=split();
391       }
392       else if(tr->bc.is_to_split())
393       {
394         new_triangles=tr->split();
395       }
396       new_triangles.push_back(tr);
397     }
398     //if bc is the longest edge
399     else if(longest_edge()==&bc)
400     {
401       //create the new triangle
402       Triangle* tr=new Triangle(ab.p1(),bc.p1(),(bc.p1()+bc.p2())/2.,
403                                                 hanging_allowed);
404       //copy information of the edge bc
405       tr->ab=ab;
406       //set my bc correctly
407       ab=ShapeEdge(ab.p1(),(bc.p1()+bc.p2())/2.);
408       //if hanging nodes are allowed transfer hanging nodes correctly
409       if(hanging_allowed)
410       {
411         std::list<Point3D> hanging_nodes=bc.get_hanging_nodes();
412         for(std::list<Point3D>::iterator i=hanging_nodes.begin();
413                                          i!=hanging_nodes.end();++i)
414         {
415           if((*i)==(tr->bc.p1()+tr->bc.p2())/2.)
416           {
417             tr->bc.add_hanging(*i);
418             bc.remove_hanging(*i);
```

```
419           }
420           if((*i)==((bc.p1()+bc.p2())/2.))
421             bc.remove_hanging(*i);
422         }
423       }
424       //set point b of my bc correctly and unmark for splitting
425       bc.set_p1((bc.p1()+bc.p2())/2.);
426       bc.unmark_to_split();
427       //now update all neighbourhood relationships
428       std::list<Shape*> old_neighbours=neighbours;
429       for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
430                                             i!=old_neighbours.end();++i)
431       {
432         (*i)->check_neighbourhood(this);
433         check_neighbourhood(*i);
434         (*i)->add_neighbour(tr);
435         tr->add_neighbour(*i);
436       }
437       add_neighbour(tr);
438       tr->add_neighbour(this);
439       //if the subtriangles need further to be split call split for the
440       //corresponding one --> note: it cant happen, that both must be
441       //split.
442       if(ca.is_to_split())
443       {
444         new_triangles=split();
445       }
446       else if(tr->ab.is_to_split())
447       {
448         new_triangles=tr->split();
449       }
450       new_triangles.push_back(tr);
451     }
452     //if ca is the longest edge
453     else if(longest_edge()==&ca)
454     {
455       //create the new triangle
456       Triangle* tr=new Triangle(ab.p2(),bc.p2(),(ca.p1()+ca.p2())/2.,
457                                                 hanging_allowed);
458       //copy information of the edge bc
459       tr->ab=bc;
460       //set my bc correctly
461       bc=ShapeEdge(ab.p2(),(ca.p1()+ca.p2())/2.);
462       //if hanging nodes are allowed transfer hanging nodes correctly
463       if(hanging_allowed)
464       {
465         std::list<Point3D> hanging_nodes=ca.get_hanging_nodes();
466         for(std::list<Point3D>::iterator i=hanging_nodes.begin();
467                                          i!=hanging_nodes.end();++i)
468         {
469           if((*i)==(tr->bc.p1()+tr->bc.p2())/2.)
470           {
471             tr->bc.add_hanging(*i);
472             ca.remove_hanging(*i);
473           }
474           if((*i)==((ca.p1()+ca.p2())/2.))
475             ca.remove_hanging(*i);
476         }
477       }
478       //set point c of my ca correctly and unmark for splitting
479       ca.set_p1((ca.p1()+ca.p2())/2.);
480       ca.unmark_to_split();
481       //now update all neighbourhood relationships
```

```
482       std::list<Shape*> old_neighbours=neighbours;
483       for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
484                                            i!=old_neighbours.end();++i)
485       {
486         (*i)->check_neighbourhood(this);
487         check_neighbourhood(*i);
488         (*i)->add_neighbour(tr);
489         tr->add_neighbour(*i);
490       }
491       add_neighbour(tr);
492       tr->add_neighbour(this);
493       //if the subtriangles need further to be split call split for the
494       //corresponding one --> note: it cant happen, that both must be
495       //split.
496       if(ab.is_to_split())
497         new_triangles=split();
498       else if(tr->ab.is_to_split())
499         new_triangles=tr->split();
500       new_triangles.push_back(tr);
501     }
502   }
503   return new_triangles;
504 }
505
506 void Triangle::listen_edge(ShapeEdge* edge)
507 {
508   ShapeEdge* myedge=0;
509   if(ab.is_masteredge_of(*edge))
510     myedge=&ab;
511   if(bc.is_masteredge_of(*edge))
512     myedge=&bc;
513   if(ca.is_masteredge_of(*edge))
514     myedge=&ca;
515   //if one of my edges is a masteredge, check for the addition of
516   //hanging nodes or for marking to split (and then also marking the
517   //longest edge).
518   if(myedge!=0 && edge->is_to_split())
519   {
520     if(hanging_allowed)
521     {
522       if(((edge->p1()+edge->p2())/2.==
523           (myedge->p1()+(myedge->p1()+myedge->p2())/2.)/2.) ||
524          ((edge->p1()+edge->p2())/2.==
525           (myedge->p2()+(myedge->p1()+myedge->p2())/2.)/2.))
526       {
527         myedge->mark_to_split();
528         if(longest_edge()!=myedge)
529         {
530           if(longest_edge()->mark_to_split())
531             longest_edge()->send();
532         }
533         if(ab.is_to_split() && bc.is_to_split() && ca.is_to_split())
534           marked_red=true;
535       }
536       else if((edge->p1()+edge->p2())/2.==(myedge->p1()+myedge->p2())/2.)
537         myedge->add_hanging((edge->p1()+edge->p2())/2.);
538     }
539     else
540     {
541       myedge->mark_to_split();
542       if(longest_edge()!=myedge && (longest_edge()->mark_to_split()))
543         longest_edge()->send();
544       if(ab.is_to_split() && bc.is_to_split() && ca.is_to_split())
```

```
545         marked_red=true;
546       }
547     }
548   }
549
550   void Triangle::check_neighbourhood(Shape* neighbour)
551   {
552     neighbours.remove(neighbour);
553     ab.remove_neighbour(neighbour);
554     bc.remove_neighbour(neighbour);
555     ca.remove_neighbour(neighbour);
556     ShapeEdge* myedge=find_neighbour_edge(neighbour);
557     if(myedge!=0)
558     {
559       neighbours.push_back(neighbour);
560       myedge->add_neighbour(neighbour);
561     }
562   }
563
564   void Triangle::display_information() const
565   {
566     std::cout<<ab.p1().x()<<",␣"<<ab.p1().y()<<",␣"<<ab.p1().z()<<"\n";
567     std::cout<<bc.p1().x()<<",␣"<<bc.p1().y()<<",␣"<<bc.p1().z()<<"\n";
568     std::cout<<ca.p1().x()<<",␣"<<ca.p1().y()<<",␣"<<ca.p1().z()<<"\n";
569     std::cout<<"I␣have␣"<<neighbours.size()<<"␣neighbours.\n";
570     std::cout<<"My␣edges␣have␣"<<ab.number_of_neighbours()<<",␣"<<bc.
        number_of_neighbours();
571     std::cout<<"␣and␣"<<ca.number_of_neighbours()<<"neighbours.\n";
572   }
573
574   std::vector<Point3D> Triangle::get_vertices() const
575   {
576     std::vector<Point3D> return_var=std::vector<Point3D>(3);
577     return_var[0]=ab.p1();return_var[1]=bc.p1();return_var[2]=ca.p1();
578     return return_var;
579   }
580
581
582   double Triangle::gram_det() const
583   {
584     std::vector<Point3D> points=get_vertices();
585     Point3D a=points[0];
586     Point3D b=points[1];
587     Point3D c=points[2];
588
589     double m11=(b.x()-a.x())*(b.x()-a.x())+(b.y()-a.y())*
590                (b.y()-a.y())+(b.z()-a.z())*(b.z()-a.z());
591     double m12=(b.x()-a.x())*(c.x()-a.x())+(b.y()-a.y())*
592                (c.y()-a.y())+(b.z()-a.z())*(c.z()-a.z());
593     double m22=(c.x()-a.x())*(c.x()-a.x())+(c.y()-a.y())*
594                (c.y()-a.y())+(c.z()-a.z())*(c.z()-a.z());
595
596     return m11*m22-m12*m12;
597   }
598
599   //************************************************************************//
600   // Implementation of functions in Rectangle                               //
601   //************************************************************************//
602   ShapeEdge* Rectangle::find_neighbour_edge(Shape* neighbour)
603   {
604     //get the edges of the neighbour
605     std::vector<ShapeEdge*> neighbour_edges=neighbour->get_edges();
606
```

```
607   if(ab.corresponds_to(*neighbour_edges[0]) ||
608      ab.corresponds_to(*neighbour_edges[1]) ||
609      ab.corresponds_to(*neighbour_edges[2]) ||
610      ab.corresponds_to(*neighbour_edges[3]))
611     return &ab;
612   if(bc.corresponds_to(*neighbour_edges[0]) ||
613      bc.corresponds_to(*neighbour_edges[1]) ||
614      bc.corresponds_to(*neighbour_edges[2]) ||
615      bc.corresponds_to(*neighbour_edges[3]))
616     return &bc;
617   if(cd.corresponds_to(*neighbour_edges[0]) ||
618      cd.corresponds_to(*neighbour_edges[1]) ||
619      cd.corresponds_to(*neighbour_edges[2]) ||
620      cd.corresponds_to(*neighbour_edges[3]))
621     return &cd;
622   if(da.corresponds_to(*neighbour_edges[0]) ||
623      da.corresponds_to(*neighbour_edges[1]) ||
624      da.corresponds_to(*neighbour_edges[2]) ||
625      da.corresponds_to(*neighbour_edges[3]))
626     return &da;
627   return 0;
628 }
629
630 Rectangle::Rectangle()
631 {
632   ab=ShapeEdge();
633   bc=ShapeEdge();
634   cd=ShapeEdge();
635   da=ShapeEdge();
636   hanging_allowed=true;
637   marked_red=false;
638   neighbours=std::list<Shape*>();
639   my_border_type=DIRICHLET;
640 }
641
642 Rectangle::Rectangle(bool hanging)
643 {
644   ab=ShapeEdge();
645   bc=ShapeEdge();
646   cd=ShapeEdge();
647   da=ShapeEdge();
648   hanging_allowed=hanging;
649   marked_red=false;
650   neighbours=std::list<Shape*>();
651   my_border_type=DIRICHLET;
652 }
653
654 Rectangle::Rectangle(Point3D a,Point3D b,Point3D c,Point3D d)
655 {
656   ab=ShapeEdge(a,b);
657   bc=ShapeEdge(b,c);
658   cd=ShapeEdge(c,d);
659   da=ShapeEdge(d,a);
660   hanging_allowed=true;
661   marked_red=false;
662   neighbours=std::list<Shape*>();
663   my_border_type=DIRICHLET;
664 }
665
666 Rectangle::Rectangle(Point3D a,Point3D b,Point3D c,Point3D d,bool hanging)
667 {
668   ab=ShapeEdge(a,b);
669   bc=ShapeEdge(b,c);
```

```
670   cd=ShapeEdge(c,d);
671   da=ShapeEdge(d,a);
672   hanging_allowed=hanging;
673   marked_red=false;
674   neighbours=std::list<Shape*>();
675   my_border_type=DIRICHLET;
676 }
677
678 Rectangle::Rectangle(Point3D a,Point3D b,Point3D c,Point3D d,
679                              border_type neumann_or_dirichlet)
680 {
681   ab=ShapeEdge(a,b);
682   bc=ShapeEdge(b,c);
683   cd=ShapeEdge(c,d);
684   da=ShapeEdge(d,a);
685   hanging_allowed=true;
686   marked_red=false;
687   neighbours=std::list<Shape*>();
688   my_border_type=neumann_or_dirichlet;
689 }
690
691 Rectangle::Rectangle(Point3D a,Point3D b,Point3D c,Point3D d,
692                      bool hanging,border_type neumann_or_dirichlet)
693 {
694   ab=ShapeEdge(a,b);
695   bc=ShapeEdge(b,c);
696   cd=ShapeEdge(c,d);
697   da=ShapeEdge(d,a);
698   hanging_allowed=hanging;
699   marked_red=false;
700   neighbours=std::list<Shape*>();
701   my_border_type=neumann_or_dirichlet;
702 }
703
704 Rectangle::Rectangle(const Rectangle& T)
705 {
706   std::vector<Point3D> points=T.get_vertices();
707   ab=ShapeEdge(points[0],points[1]);
708   bc=ShapeEdge(points[1],points[2]);
709   cd=ShapeEdge(points[2],points[3]);
710   da=ShapeEdge(points[3],points[0]);
711   hanging_allowed=T.is_hanging_allowed();
712   marked_red=T.is_to_split();
713   neighbours=std::list<Shape*>();
714   my_border_type=T.border_information();
715 }
716
717 Rectangle::~Rectangle()
718 {
719   ab.~ShapeEdge();
720   bc.~ShapeEdge();
721   cd.~ShapeEdge();
722   da.~ShapeEdge();
723   neighbours.~list<Shape*>();
724 }
725
726 std::vector<Point3D> Rectangle::get_vertices() const
727 {
728   std::vector<Point3D> return_var=std::vector<Point3D>(4);
729   return_var[0]=ab.p1();
730   return_var[1]=bc.p1();
731   return_var[2]=cd.p1();
732   return_var[3]=da.p1();
```

```
733   return return_var;
734 }
735
736
737 std::vector<ShapeEdge*> Rectangle::get_edges()
738 {
739   std::vector<ShapeEdge*> edges=std::vector<ShapeEdge*>(4);
740   edges[0]=&ab;
741   edges[1]=&bc;
742   edges[2]=&cd;
743   edges[3]=&da;
744   return edges;
745 }
746
747 std::vector<Shape*> Rectangle::get_neighbours()
748 {
749   std::vector<Shape*> n=std::vector<Shape*>();
750   for(std::list<Shape*>::iterator i=neighbours.begin();i!=neighbours.end();++i)
751     n.push_back(*i);
752   return n;
753 }
754
755 void Rectangle::add_neighbour(Shape* neighbour)
756 {
757   ShapeEdge* edge=find_neighbour_edge(neighbour);
758   if(edge!=0)
759   {
760     edge->add_neighbour(neighbour);
761     //add the neighbour to the list, if its not yet listed
762     if(find(neighbours.begin(),neighbours.end(),neighbour)==neighbours.end())
763     {
764       neighbours.push_back(neighbour);
765     }
766   }
767 }
768
769 void Rectangle::remove_neighbour(Shape* neighbour)
770 {
771   neighbours.remove(neighbour);
772   ab.remove_neighbour(neighbour);
773   bc.remove_neighbour(neighbour);
774   cd.remove_neighbour(neighbour);
775   da.remove_neighbour(neighbour);
776 }
777
778 void Rectangle::mark_red(bool contact_neighbours)
779 {
780   if(ab.mark_to_split() && contact_neighbours)
781     ab.send();
782   if(bc.mark_to_split() && contact_neighbours)
783     bc.send();
784   if(cd.mark_to_split() && contact_neighbours)
785     cd.send();
786   if(da.mark_to_split() && contact_neighbours)
787     da.send();
788   marked_red=true;
789 }
790
791 void Rectangle::mark_red_x(bool contact_neighbours)
792 {
793   if(ab.mark_to_split() && contact_neighbours)
794     ab.send();
795   if(cd.mark_to_split() && contact_neighbours)
```

```
796       cd.send();
797     marked_red=true;
798   }
799
800   void Rectangle::mark_red_y(bool contact_neighbours)
801   {
802     if(bc.mark_to_split() && contact_neighbours)
803       bc.send();
804     if(da.mark_to_split() && contact_neighbours)
805       da.send();
806     marked_red=true;
807   }
808
809
810   bool Rectangle::is_to_split() const
811   {return marked_red;}
812
813   std::vector<Shape*> Rectangle::split(bool consider_neighbourhood)
814   {
815     std::vector<Shape*> new_rectangles;//return variable
816     marked_red=false;//the modified rectangle should not be split
817     //create the new rectangles
818     if(ab.is_to_split() && da.is_to_split())
819     {
820       new_rectangles=std::vector<Shape*>(3);
821       Point3D center_point=((ab.p1()+ab.p2())/2.+(cd.p1()+cd.p2())/2.)/2.;
822       Rectangle* rec1=new Rectangle((ab.p1()+ab.p2())/2.,ab.p2(),
823                                     (bc.p1()+bc.p2())/2.,
824                                     center_point,hanging_allowed);
825       Rectangle* rec2=new Rectangle(center_point,(bc.p1()+bc.p2())/2.,bc.p2(),
826                                     (cd.p1()+cd.p2())/2.,hanging_allowed);
827       Rectangle* rec3=new Rectangle((da.p1()+da.p2())/2.,center_point,
828                                     (cd.p1()+cd.p2())/2.,
829                                     da.p1(),hanging_allowed);
830       if(hanging_allowed)//if there might be hanging nodes
831       {//add the hanging nodes of the first edge
832         std::list<Point3D> hanging_nodes=ab.get_hanging_nodes();
833         std::vector<ShapeEdge*> edges1=rec1->get_edges();
834         std::list<Point3D>::const_iterator i;
835         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
836         {
837           edges1[0]->add_hanging(*i);
838           if(!((*i)==((ab.p1()+ab.p2())/2.+ab.p1())/2.))
839             ab.remove_hanging(*i);
840         }
841         //add the hanging nodes of the second edge
842         hanging_nodes=bc.get_hanging_nodes();
843         std::vector<ShapeEdge*> edges2=rec2->get_edges();
844         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
845         {
846           edges1[1]->add_hanging(*i);
847           edges2[1]->add_hanging(*i);
848         }
849         //add the hanging nodes of the third edge
850         hanging_nodes=cd.get_hanging_nodes();
851         edges1=rec2->get_edges();
852         edges2=rec3->get_edges();
853         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
854         {
855           edges1[2]->add_hanging(*i);
856           edges2[2]->add_hanging(*i);
857         }
858         //add the hanging nodes of the fourth edge
```

```
859         hanging_nodes=da.get_hanging_nodes();
860         edges1=rec3->get_edges();
861         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
862         {
863           edges1[3]->add_hanging(*i);
864           if(!((*i)==((da.p1()+da.p2())/2.+da.p1())/2.))
865             da.remove_hanging(*i);
866         }
867       }
868       //correct my own edges
869       ab.set_p2((ab.p1()+ab.p2())/2.);
870       ab.unmark_to_split();
871       da.set_p1((da.p1()+da.p2())/2.);
872       da.unmark_to_split();
873       bc=ShapeEdge(ab.p2(),center_point);
874       cd=ShapeEdge(center_point,da.p1());
875       if(consider_neighbourhood)
876       {
877         //set neighbourhood-relationships
878         std::list<Shape*> old_neighbours=neighbours;
879         for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
880                                               i!=old_neighbours.end();++i)
881         {
882           rec1->add_neighbour(*i);//Add my neighbours to the new triangles
883           rec2->add_neighbour(*i);
884           rec3->add_neighbour(*i);
885           (*i)->check_neighbourhood(this);//eventually remove myself from neighbours
886           (*i)->add_neighbour(rec1);//add the new triangles as neighbours to the
887           (*i)->add_neighbour(rec2);
888           (*i)->add_neighbour(rec3);
889           check_neighbourhood(*i);//eventually remove neighbour from my list
890         }
891         //add the relationships between the new triangles.
892         rec1->add_neighbour(this);
893         rec1->add_neighbour(rec2);
894         rec3->add_neighbour(this);
895         rec3->add_neighbour(rec2);
896         rec2->add_neighbour(rec1);
897         rec2->add_neighbour(rec3);
898
899         bc.add_neighbour(rec1);//add the new triangles to my neighbourlist
900         neighbours.push_back(rec1);//and the list of the corresponding edge
901         cd.add_neighbour(rec3);
902         neighbours.push_back(rec3);
903         rec1->add_neighbour(this);//add myself as neighbour to the new
904         rec3->add_neighbour(this);
905       }
906       //set the return variable
907       new_rectangles[0]=rec1;
908       new_rectangles[1]=rec2;
909       new_rectangles[2]=rec3;
910       return new_rectangles;
911     }
912     else if(ab.is_to_split())
913     {
914       new_rectangles=std::vector<Shape*>(1);
915       Rectangle* rec=new Rectangle((ab.p1()+ab.p2())/2.,ab.p2(),bc.p2(),
916                                    (cd.p1()+cd.p2())/2.,hanging_allowed);
917       if(hanging_allowed)//if there might be hanging nodes
918       {//add the hanging nodes of the first edge
919         std::list<Point3D> hanging_nodes=ab.get_hanging_nodes();
920         std::vector<ShapeEdge*> edges1=rec->get_edges();
921         std::list<Point3D>::const_iterator i;
```

```
922       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
923       {
924         edges1[0]->add_hanging(*i);
925         if(!((*i)==((ab.p1()+ab.p2())/2.+ab.p1())/2.))
926           ab.remove_hanging(*i);
927       }
928       //add the hanging nodes of the second edge
929       hanging_nodes=bc.get_hanging_nodes();
930       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
931         edges1[1]->add_hanging(*i);
932       //add the hanging nodes of the third edge
933       hanging_nodes=cd.get_hanging_nodes();
934       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
935       {
936         edges1[2]->add_hanging(*i);
937         if(!((*i)==((cd.p1()+cd.p2())/2.+cd.p2())/2.))
938           cd.remove_hanging(*i);
939       }
940       //add the hanging nodes of the fourth edge
941       hanging_nodes=da.get_hanging_nodes();
942       edges1=rec->get_edges();
943       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
944         edges1[3]->add_hanging(*i);
945     }
946     //correct my own edges
947     ab.set_p2((ab.p1()+ab.p2())/2.);
948     ab.unmark_to_split();
949     cd.set_p1((cd.p1()+cd.p2())/2.);
950     cd.unmark_to_split();
951     bc=ShapeEdge(ab.p2(),cd.p1());
952     if(consider_neighbourhood)
953     {
954       //set neighbourhood-relationships
955       std::list<Shape*> old_neighbours=neighbours;
956       for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
957                                             i!=old_neighbours.end();++i)
958       {
959         rec->add_neighbour(*i);//Add my neighbours to the new triangles
960         (*i)->check_neighbourhood(this);//eventually remove myself from neighbours
961         (*i)->add_neighbour(rec);//add the new triangles as neighbours to the
962         check_neighbourhood(*i);//eventually remove neighbour from my list
963       }
964
965       rec->add_neighbour(this);
966
967       bc.add_neighbour(rec);//add the new triangles to my neighbourlist
968       neighbours.push_back(rec);//and the list of the corresponding edge
969     }
970     //set the return variable
971     new_rectangles[0]=rec;
972     return new_rectangles;
973   }
974   else if(da.is_to_split())
975   {
976     new_rectangles=std::vector<Shape*>(1);
977     Rectangle* rec=new Rectangle((da.p1()+da.p2())/2.,da.p2(),ab.p2(),
978                                  (bc.p1()+bc.p2())/2.,hanging_allowed);
979     if(hanging_allowed)//if there might be hanging nodes
980     {//add the hanging nodes of the first edge
981       std::list<Point3D> hanging_nodes=ab.get_hanging_nodes();
982       std::vector<ShapeEdge*> edges1=rec->get_edges();
983       std::list<Point3D>::const_iterator i;
984       for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
```

```
985           edges1[0]->add_hanging(*i);
986         //add the hanging nodes of the second edge
987         hanging_nodes=bc.get_hanging_nodes();
988         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
989         {
990           edges1[1]->add_hanging(*i);
991           if(!((*i)==((bc.p1()+bc.p2())/2.+bc.p2())/2.))
992             da.remove_hanging(*i);
993         }
994         //add the hanging nodes of the third edge
995         hanging_nodes=cd.get_hanging_nodes();
996         for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
997           edges1[2]->add_hanging(*i);
998         //add the hanging nodes of the fourth edge
999         hanging_nodes=da.get_hanging_nodes();
1000        edges1=rec->get_edges();
1001        for(i=hanging_nodes.begin();i!=hanging_nodes.end();++i)
1002        {
1003          edges1[3]->add_hanging(*i);
1004          if(!((*i)==((da.p1()+da.p2())/2.+da.p1())/2.))
1005            da.remove_hanging(*i);
1006        }
1007      }
1008      //correct my own edges
1009      bc.set_p1((bc.p1()+bc.p2())/2.);
1010      bc.unmark_to_split();
1011      da.set_p2((da.p1()+da.p2())/2.);
1012      da.unmark_to_split();
1013      ab=ShapeEdge(da.p2(),bc.p1());
1014      if(consider_neighbourhood)
1015      {
1016        //set neighbourhood-relationships
1017        std::list<Shape*> old_neighbours=neighbours;
1018        for(std::list<Shape*>::const_iterator i=old_neighbours.begin();
1019                                              i!=old_neighbours.end();++i)
1020        {
1021          rec->add_neighbour(*i);//Add my neighbours to the new triangles
1022          (*i)->check_neighbourhood(this);//eventually remove myself from neighbours
1023          (*i)->add_neighbour(rec);//add the new triangles as neighbours to the
1024          check_neighbourhood(*i);//eventually remove neighbour from my list
1025        }
1026
1027        rec->add_neighbour(this);
1028
1029        ab.add_neighbour(rec);//add the new triangles to my neighbourlist
1030        neighbours.push_back(rec);//and the list of the corresponding edge
1031      }
1032      //set the return variable
1033      new_rectangles[0]=rec;
1034      return new_rectangles;
1035    }
1036    else
1037      return new_rectangles;
1038  }
1039
1040  void Rectangle::allow_hanging()
1041  {hanging_allowed=true;}
1042
1043  void Rectangle::listen_edge(ShapeEdge* edge)
1044  {
1045    ShapeEdge* myedge=0;
1046    if(ab.is_masteredge_of(*edge))
1047      myedge=&ab;
```

```
1048    else if(bc.is_masteredge_of(*edge))
1049      myedge=&bc;
1050    else if(cd.is_masteredge_of(*edge))
1051      myedge=&cd;
1052    else if(da.is_masteredge_of(*edge))
1053      myedge=&da;
1054    //if one of my edges is a masteredge, check for the addition of
1055    //hanging nodes or for marking to split (and then also marking the
1056    //longest edge).
1057    if(myedge!=0 && edge->is_to_split())
1058    {
1059      if(hanging_allowed)
1060      {
1061        if(((edge->p1()+edge->p2())/2.==
1062           (myedge->p1()+(myedge->p1()+myedge->p2())/2.)/2.) ||
1063           ((edge->p1()+edge->p2())/2.==
1064           (myedge->p2()+(myedge->p1()+myedge->p2())/2.)/2.))
1065        {
1066          marked_red=true;
1067          if(myedge==&ab || myedge==&cd)
1068          {
1069            if(ab.mark_to_split() && &ab!=myedge)
1070              ab.send();
1071            if(cd.mark_to_split() && &cd!=myedge)
1072              cd.send();
1073          }
1074          if(myedge==&bc || myedge==&da)
1075          {
1076            if(bc.mark_to_split() && &bc!=myedge)
1077              bc.send();
1078            if(da.mark_to_split() && &da!=myedge)
1079              da.send();
1080          }
1081        }
1082        else if((edge->p1()+edge->p2())/2.==(myedge->p1()+myedge->p2())/2.)
1083          myedge->add_hanging((edge->p1()+edge->p2())/2.);
1084      }
1085      else
1086      {
1087        marked_red=true;
1088        if(myedge==&ab || myedge==&cd)
1089        {
1090          if(ab.mark_to_split() && &ab!=myedge)
1091            ab.send();
1092          if(bc.mark_to_split() && &bc!=myedge)
1093            bc.send();
1094        }
1095        if(myedge==&bc || myedge==&da)
1096        {
1097          if(bc.mark_to_split() && &bc!=myedge)
1098            bc.send();
1099          if(da.mark_to_split() && &da!=myedge)
1100            da.send();
1101        }
1102      }
1103    }
1104  }
1105
1106
1107  void Rectangle::check_neighbourhood(Shape* neighbour)
1108  {
1109    neighbours.remove(neighbour);
1110    ab.remove_neighbour(neighbour);
```

```
1111    bc.remove_neighbour(neighbour);
1112    cd.remove_neighbour(neighbour);
1113    da.remove_neighbour(neighbour);
1114    ShapeEdge* myedge=find_neighbour_edge(neighbour);
1115    if(myedge!=0)
1116    {
1117      neighbours.push_back(neighbour);
1118      myedge->add_neighbour(neighbour);
1119    }
1120  }
1121
1122
1123  void Rectangle::display_information() const
1124  {
1125    std::cout<<ab.p1().x()<<",␣"<<ab.p1().y()<<",␣"<<ab.p1().z()<<"\n";
1126    std::cout<<bc.p1().x()<<",␣"<<bc.p1().y()<<",␣"<<bc.p1().z()<<"\n";
1127    std::cout<<cd.p1().x()<<",␣"<<cd.p1().y()<<",␣"<<cd.p1().z()<<"\n";
1128    std::cout<<da.p1().x()<<",␣"<<da.p1().y()<<",␣"<<da.p1().z()<<"\n";
1129    std::cout<<"I␣have␣"<<neighbours.size()<<"␣neighbours.\n";
1130    std::cout<<"My␣edges␣have␣"<<ab.number_of_neighbours()<<",␣"<<
1131               bc.number_of_neighbours();
1132    std::cout<<",␣"<<cd.number_of_neighbours()<<"␣and␣"<<da.number_of_neighbours()
1133        <<"neighbours.\n";
1134  }
1135
1136  border_type Rectangle::border_information() const
1137  {return my_border_type;}
1138
1139  //NOTE! For Rectnagles gram_det() simply returns the area! It is assumed,
1140  //that the Rectangle is axis-parallel and should better not be degenerated.
1141  double Rectangle::gram_det() const
1142  {
1143    std::vector<Point3D> points=get_vertices();
1144    Point3D a=points[0];
1145    Point3D b=points[1];
1146    Point3D c=points[2];
1147    Point3D d=points[3];
1148
1149    if(a.x()==c.x())
1150    {
1151      double ymin=a.y();
1152      double ymax=a.y();
1153      double zmin=a.z();
1154      double zmax=a.z();
1155      for(int i=1;i<4;++i)
1156      {
1157        if(points[i].y()<ymin)
1158          ymin=points[i].y();
1159        if(points[i].y()>ymax)
1160          ymax=points[i].y();
1161        if(points[i].z()<zmin)
1162          zmin=points[i].z();
1163        if(points[i].z()>zmax)
1164          zmax=points[i].z();
1165      }
1166      return (ymax-ymin)*(zmax-zmin);
1167    }
1168    if(a.y()==c.y())
1169    {
1170      double ymin=a.x();
1171      double ymax=a.x();
1172      double zmin=a.z();
1173      double zmax=a.z();
```

```
1174      for(int i=1;i<4;++i)
1175      {
1176        if(points[i].x()<ymin)
1177          ymin=points[i].x();
1178        if(points[i].x()>ymax)
1179          ymax=points[i].x();
1180        if(points[i].z()<zmin)
1181          zmin=points[i].z();
1182        if(points[i].z()>zmax)
1183          zmax=points[i].z();
1184      }
1185      return (ymax-ymin)*(zmax-zmin);
1186    }
1187    if(a.z()==c.z())
1188    {
1189      double ymin=a.x();
1190      double ymax=a.x();
1191      double zmin=a.y();
1192      double zmax=a.y();
1193      for(int i=1;i<4;++i)
1194      {
1195        if(points[i].x()<ymin)
1196          ymin=points[i].x();
1197        if(points[i].x()>ymax)
1198          ymax=points[i].x();
1199        if(points[i].y()<zmin)
1200          zmin=points[i].y();
1201        if(points[i].y()>zmax)
1202          zmax=points[i].y();
1203      }
1204      return (ymax-ymin)*(zmax-zmin);
1205    }
1206
1207    return 0.;
1208  }
1209
1210
1211
1212  //************************************************************************//
1213  // Implementation of functions in Triangulation3D                         //
1214  //************************************************************************//
1215  Triangulation3D::Triangulation3D(){}
1216
1217  Triangulation3D::Triangulation3D(std::vector<Shape*> initial_shapes)
1218  {
1219    consider_neighbourhood=true;
1220    reserve(initial_shapes.size());
1221    int j=0;
1222    for(int i=0;i<initial_shapes.size();++i)
1223      if(find(begin(),end(),initial_shapes[i])==end())
1224        push_back(initial_shapes[i]);
1225    data=NULL;
1226  }
1227
1228  Triangulation3D::Triangulation3D(std::vector<Shape*> initial_shapes,
1229                                                    bool set_neighbours)
1230  {
1231    consider_neighbourhood=set_neighbours;
1232    reserve(initial_shapes.size());
1233    int j=0;
1234    for(int i=0;i<initial_shapes.size();++i)
1235      if(find(begin(),end(),initial_shapes[i])==end())
1236        push_back(initial_shapes[i]);
```

```
1237   data=NULL;
1238 }
1239
1240 Triangulation3D::Triangulation3D(std::string filename)
1241 {
1242   std::cout<<"Sorry␣this␣form␣of␣mesh-creation␣is␣not␣yet␣implemented!\n";
1243 }
1244
1245 Triangulation3D::Triangulation3D(const Triangulation3D& T)
1246 {
1247   consider_neighbourhood=T.is_considering_neighbourhood();
1248   reserve(T.size());
1249   if(T[0]->what_are_you()==TRIANGLE)
1250   {
1251     Triangle* Tr=NULL;
1252     for(int i=0;i<T.size();++i)
1253     {
1254       std::vector<Point3D> points=T[i]->get_vertices();
1255       Tr=new Triangle(points[0],points[1],points[2]);
1256       push_back(Tr);
1257     }
1258   }
1259   else
1260   {
1261     Rectangle* Tr=NULL;
1262     for(int i=0;i<T.size();++i)
1263     {
1264       std::vector<Point3D> points=T[i]->get_vertices();
1265       Tr=new Rectangle(points[0],points[1],points[2],points[3]);
1266       push_back(Tr);
1267     }
1268   }
1269   //check all neighbourhood relations (slow operation)
1270   for(int i=0;i<size();++i)
1271     for(int j=i+1;j<size();++j)
1272       (*this)[i]->add_neighbour((*this)[j]);
1273   data = T.data;
1274 }
1275
1276 Triangulation3D::~Triangulation3D()
1277 {
1278   for(int i=0;i<size();++i)
1279     delete (*this)[i];
1280 }
1281
1282 Triangulation3D& Triangulation3D::operator=(const Triangulation3D& T)
1283 {
1284   if(this!=&T)
1285   {
1286     consider_neighbourhood=T.is_considering_neighbourhood();
1287     clear();
1288     reserve(T.size());
1289     if(T[0]->what_are_you()==TRIANGLE)
1290     {
1291       Triangle* Tr=NULL;
1292       for(int i=0;i<T.size();++i)
1293       {
1294         Tr=new Triangle(T[i]);
1295         push_back(Tr);
1296       }
1297     }
1298     else
1299     {
```

```
1300        Rectangle* Tr=NULL;
1301        for(int i=0;i<T.size();++i)
1302        {
1303          Tr=new Rectangle(T[i]);
1304          push_back(Tr);
1305        }
1306      }
1307      //check all neighbourhood relations (slow operation)
1308      for(int i=0;i<size();++i)
1309        for(int j=i+1;j<size();++j)
1310          (*this)[i]->add_neighbour((*this)[j]);
1311    }
1312    data=T.data;
1313  }
1314
1315  bool Triangulation3D::is_considering_neighbourhood() const
1316  {
1317    return consider_neighbourhood;
1318  }
1319
1320  void Triangulation3D::regulate_diameters(double k)
1321  {
1322    bool is_regular=false;
1323    while(!is_regular)
1324    {
1325      is_regular=true;
1326      for(int i=0;i<size();++i)
1327      {
1328        std::vector<Point3D> v=(*this)[i]->get_vertices();
1329        double h=v[0].distance(v[1]);
1330        if(h<v[1].distance(v[2]))
1331          h=v[1].distance(v[2]);
1332        std::vector<Shape*> n=(*this)[i]->get_neighbours();
1333        for(int j=0;j<n.size();++j)
1334        {
1335          v=n[j]->get_vertices();
1336          double h_n=v[0].distance(v[1]);
1337          if(h_n<v[1].distance(v[2]))
1338            h_n=v[1].distance(v[2]);
1339          if(k*h_n<h)
1340          {
1341            v=(*this)[i]->get_vertices();
1342            if(v[0].distance(v[1])>v[1].distance(v[2]))
1343              (*this)[i]->mark_red_x();
1344            else if(v[0].distance(v[1])<v[1].distance(v[2]))
1345              (*this)[i]->mark_red_y();
1346            else
1347              (*this)[i]->mark_red();
1348            is_regular=false;
1349          }
1350        }
1351      }
1352      if(!is_regular)
1353        refine();
1354    }
1355  }
1356
1357  void Triangulation3D::refine()
1358  {
1359    std::vector<Shape*> new_shapes=std::vector<Shape*>();
1360    for(int i=0;i<size();++i)
1361      if(((*this)[i])->is_to_split())
1362      {
```

```
1363        std::vector<Shape*> very_new_shapes=
1364                            (*this)[i]->split(consider_neighbourhood);
1365        for(int j=0;j<very_new_shapes.size();++j)
1366          new_shapes.push_back(very_new_shapes[j]);
1367      }
1368    reserve(size()+new_shapes.size());
1369    for(int i=0;i<new_shapes.size();++i)
1370      push_back(new_shapes[i]);
1371  }
1372
1373  void Triangulation3D::refine_uniformily()
1374  {
1375    std::vector<Shape*> new_shapes=std::vector<Shape*>();
1376    for(int i=0;i<size();++i)
1377    {
1378      (*this)[i]->mark_red();
1379    }
1380    for(int i=0;i<size();++i)
1381    {
1382      std::vector<Shape*> very_new_shapes=
1383                          (*this)[i]->split(consider_neighbourhood);
1384      for(int j=0;j<very_new_shapes.size();++j)
1385        new_shapes.push_back(very_new_shapes[j]);
1386    }
1387    reserve(size()+new_shapes.size());
1388    for(int i=0;i<new_shapes.size();++i)
1389      push_back(new_shapes[i]);
1390  }
1391
1392
1393  }
```

A.3.5 triangulationio.h

```
1  //************************************************************************//
2  //File:                    triangulationio.h                              //
3  //This is the header file for the input output routines concerning all classes//
4  //of the tr3D namespace as specified in triangulation.h These are pure    //
5  //algorithms most of them used to write or read triangulation data to or from //
6  //files specified by the user. It specially contains functions to write and  //
7  //read the triangulation-data in the hlib native format.                  //
8  //************************************************************************//
9  //Author: Samuel Ferraz-Leite                                             //
10 //Date: 07.2007                                                           //
11 //Version 0.8 mot all features implemented                               //
12 //************************************************************************//
13 #ifndef BEMLIB_TRIANGULATIONIO_H
14 #define BEMLIB_TRIANGULATIONIO_H
15
16 #define HAVE_HLIB
17 #ifdef HAVE_HLIB
18 //Include the main header from HLib to use for BEM
19 extern "C"
20 {
21 #include "bem3d.h"
22 }
23 #endif
24
25 //Include FLBEM-modules
26 #include "triangulation.h"
27
28
29 namespace triangulation{
30
```

```
31 //This is the routine to write a Triangulation3D object to the
32 //native hlib file format. It is assumed, that a Trianguoltion3D
33 //containes Triangles! hlib has no support for rectangles!
34 void write_hlib_bemgrid3D(const char*,const Triangulation3D&);
35
36 //This routine draws the projection of a Triangulation3D
37 //as specified in triangulation.h onto the xy-plane. Data is stored
38 //in postscript format.
39 void draw_tr3d_projection_to_xy(const char*,const Triangulation3D&);
40
41 //This routine writes the data of a Triangulation3D object onto
42 //two files, such that it can be easily read by matlab routines.
43 void write_matlab_tr3d(const char*,const char*,const Triangulation3D&);
44
45 //This routine writes the x-Koordinates, the y-Koordinates and the
46 //z_Koordinates of a Mesh into fiels to be used by tr3d_visualize.m
47 void write_matlab_visualisation_data_tr3d(const char*,const char*,const char*,
48                                                    const Triangulation3D&);
49
50 #ifdef HAVE_HLIB
51 //create a mesh in the hlib format from an existing triangulation3d.
52 //Note! It is assumed, that the triangulation3d contains only triagles
53 pbemgrid3d triangulation3d_to_bemgrid3d(const Triangulation3D&);
54 #endif
55
56 //Read a graded Mesh from two Datafiles. These routines work with the Data-
57 //files that were kindly provided by Stefan Funken
58 Triangulation3D create_graded_mesh(const char* koords,const char* elements);
59 Triangulation3D create_graded_cube_from_square(const char* koords,
60                                                const char* elements);
61
62 //Read a Mesh from the coordinates dat files as generated by
63 //write_matlab_visualisation_data_tr3d(). Note, that this routine is very slow,
64 //since it contains a lot of I/O and the neighbourhoods have to be set manually
65 //causing quadratic runtime behaviour! This routine has so far only been
66 //implemented for rectangles.
67 Triangulation3D create_mesh_from_visualisationdata(const char* xdat,const char* ydat
    ,
68                                                    const char* zdat,
69                                                    bool allow_hanging=true,
70                                                    bool consider_neighbours=true);
71
72 }
73 #endif
```

A.3.6 triangulationio.cpp

```
1  //*************************************************************************//
2  //File:                  triangulationio.cpp                                //
3  //This file implements all routines specified in triangulationio.h          //
4  //*************************************************************************//
5  //Author: Samuel Ferraz-Leite                                               //
6  //Date: 07.2007                                                             //
7  //Version 0.8 not all features implemented                                  //
8  //*************************************************************************//
9
10 //Includes of the FLBEM-modules
11 #include "triangulation.h"
12 #include "triangulationio.h"
13
14 //Include of the HLib-module
15 extern "C"
16 {
17 #include "bem3d.h"
```

```
18 }
19
20 //Include of the standard-libraries
21 #include<fstream>
22 #include<iostream>
23 #include<vector>
24 #include<map>
25 #include<set>
26 #include<cmath>
27
28 namespace triangulation{
29
30 void write_hlib_bemgrid3D(const char* filename,const Triangulation3D& tr_to_write)
31 {
32   std::ofstream outfile(filename);
33   if(!outfile)
34   {
35     std::cerr<<"Could not open file "<<filename;
36     return;
37   }
38
39   std::vector<Point3D*> vertices=std::vector<Point3D*>();
40   std::map<Point3D,int> vertices_index=std::map<Point3D,int>();
41   std::vector<Edge*> edges=std::vector<Edge*>();
42   std::map<Edge,int> edges_index=std::map<Edge,int>();
43
44   //Gather all the information from the Triangulation3D specially
45   //assigning and determining the index relations between vertices,
46   //edges and triangles
47   for(Triangulation3D::const_iterator i=tr_to_write.begin();
48       i!=tr_to_write.end();++i)
49   {
50     static std::pair<std::map<Point3D,int>::iterator,bool> insert_info_v;
51     static std::pair<std::map<Edge,int>::iterator,bool> insert_info_e;
52     static std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
53     edges_of_triangle=(*i)->get_edges();
54
55     //if a new vertice is found - insert with index value
56     insert_info_v=vertices_index.insert(
57         std::pair<Point3D,int>(edges_of_triangle[0]->p1(),vertices_index.size()));
58     if(insert_info_v.second)
59       vertices.push_back((Point3D*)&(insert_info_v.first->first));
60     insert_info_v=vertices_index.insert(
61         std::pair<Point3D,int>(edges_of_triangle[1]->p1(),vertices_index.size()));
62     if(insert_info_v.second)
63       vertices.push_back((Point3D*)&(insert_info_v.first->first));
64     insert_info_v=vertices_index.insert(
65         std::pair<Point3D,int>(edges_of_triangle[2]->p1(),vertices_index.size()));
66     if(insert_info_v.second)
67       vertices.push_back((Point3D*)&(insert_info_v.first->first));
68
69     //if a new edge is found - insert with index value
70     insert_info_e=edges_index.insert(
71         std::pair<Edge,int>(*(edges_of_triangle[0]),edges.size()));
72     if(insert_info_e.second)
73       edges.push_back((Edge*)&(insert_info_e.first->first));
74     insert_info_e=edges_index.insert(
75         std::pair<Edge,int>(*(edges_of_triangle[1]),edges.size()));
76     if(insert_info_e.second)
77       edges.push_back((Edge*)&(insert_info_e.first->first));
78     insert_info_e=edges_index.insert(
79         std::pair<Edge,int>(*(edges_of_triangle[2]),edges.size()));
80     if(insert_info_e.second)
```

```
81        edges.push_back((Edge*)&(insert_info_e.first->first));
82    }
83    //write all the info to the specified file.
84    outfile<<vertices.size()<<"␣"<<edges.size()<<"␣"<<tr_to_write.size()<<"\n";
85    for(int i=0;i<vertices.size();++i)
86    {
87      outfile<<vertices[i]->x()<<"␣"<<vertices[i]->y()<<"␣"
88             <<vertices[i]->z()<<"\n";
89    }
90    for(int i=0;i<edges.size();++i)
91      outfile<<vertices_index[edges[i]->p1()]<<"␣"
92             <<vertices_index[edges[i]->p2()]<<"\n";
93    for(Triangulation3D::const_iterator i=tr_to_write.begin();
94        i!=tr_to_write.end();++i)
95    {
96      static std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
97      edges_of_triangle=(*i)->get_edges();
98      outfile<<edges_index[*edges_of_triangle[0]]<<"␣";
99      outfile<<edges_index[*edges_of_triangle[1]]<<"␣";
100     outfile<<edges_index[*edges_of_triangle[2]]<<"␣";
101     outfile<<vertices_index[edges_of_triangle[0]->p1()]<<"␣";
102     outfile<<vertices_index[edges_of_triangle[1]->p1()]<<"␣";
103     outfile<<vertices_index[edges_of_triangle[2]->p1()]<<"\n";
104   }
105 }
106
107
108 void
109 draw_tr3d_projection_to_xy(const char* filename,const Triangulation3D& tr_to_draw)
110 {
111   std::ofstream outfile(filename);
112   if(!outfile)
113   {
114     std::cerr<<"Could␣not␣open␣file␣"<<filename;
115     return;
116   }
117   const int offset = 50;
118   std::set<Edge> edges=std::set<Edge>();
119   std::set<Point3D> vertices=std::set<Point3D>();
120   for(Triangulation3D::const_iterator i=tr_to_draw.begin();
121       i!=tr_to_draw.end();++i)
122   {
123     static std::vector<ShapeEdge*> edges_of_shape=std::vector<ShapeEdge*>();
124     edges_of_shape=((*i)->get_edges());
125     for(std::vector<ShapeEdge*>::const_iterator j=edges_of_shape.begin();
126         j!=edges_of_shape.end();++j)
127     {
128       edges.insert(*(*j));
129       vertices.insert((*j)->p1());
130     }
131   }
132   std::set<Edge>::const_iterator i=edges.begin();
133   double hmin=(((Edge)(*i)).p2().x()-((Edge)(*i)).p1().x())*
134         (((Edge)(*i)).p2().x()-((Edge)(*i)).p1().x())+
135         (((Edge)(*i)).p2().y()-((Edge)(*i)).p1().y())*
136         (((Edge)(*i)).p2().y()-((Edge)(*i)).p1().y());
137   double hmax=hmin;
138   double h=0.;
139   for(i=edges.begin();i!=edges.end();++i)
140   {
141     h=(((Edge)(*i)).p2().x()-((Edge)(*i)).p1().x())*
142         (((Edge)(*i)).p2().x()-((Edge)(*i)).p1().x())+
143         (((Edge)(*i)).p2().y()-((Edge)(*i)).p1().y())*
```

```
144           (((Edge)(*i)).p2().y()-((Edge)(*i)).p1().y());
145       if(h<hmin)
146         hmin=h;
147       if(h>hmax)
148         hmax=h;
149     }
150     hmin=sqrt(hmin);
151     hmax=sqrt(hmax);
152     double radius=0.1*hmin;
153 
154 
155     double bmin[2];
156     double bmax[2];
157     std::set<Point3D>::const_iterator j=vertices.begin();
158     bmin[0]=bmax[0]=((Point3D)(*j)).x();
159     bmin[1]=bmax[1]=((Point3D)(*j)).y();
160     for(j=vertices.begin();j!=vertices.end();++j)
161     {
162       if(((Point3D)(*j)).x()<bmin[0])
163         bmin[0]=((Point3D)(*j)).x();
164       else if(((Point3D)(*j)).x()>bmax[0])
165         bmax[0]=((Point3D)(*j)).x();
166 
167       if(((Point3D)(*j)).y()<bmin[1])
168         bmin[1]=((Point3D)(*j)).y();
169       else if(((Point3D)(*j)).y()>bmax[1])
170         bmax[1]=((Point3D)(*j)).y();
171     }
172 
173     bmin[0]-=radius+hmin;
174     bmin[1]-=radius+hmin;
175     bmax[0]+=radius+hmin;
176     bmax[1]+=radius+hmin;
177 
178     double scale=0;
179     if(bmax[0]-bmin[0]<bmax[1]-bmin[1])
180       scale=500./(bmax[1]-bmin[1]);
181     else
182       scale=500./(bmax[0]-bmin[0]);
183 
184     outfile<<"%!PS-Adobe-2.0␣EPSF-2.0\n%%BoundingBox:␣"<<offset<<"␣"<<offset<<"␣"
185         <<offset+(int)(scale*(bmax[0]-bmin[0])+0.5)<<"␣"
186         <<offset+(int)(scale*(bmax[1]-bmin[1])+0.5)<<"\n"
187         <<offset-scale*bmin[0]<<"␣"<<offset-scale*bmin[1]<<"␣translate\n"
188         <<scale<<"␣dup␣scale\n"
189         <<"/Helvetica␣findfont␣"<<0.4*hmin<<"␣scalefont␣setfont\n"
190         <<1.0/scale<<"␣setlinewidth\n"
191         <<"/dl{moveto␣lineto␣stroke}␣def\n"
192         <<"/dn{moveto␣dup␣stringwidth␣-0.5␣mul␣exch␣-0.5␣mul␣exch␣rmoveto␣show}␣def\n\
              n";
193 
194     for(i=edges.begin();i!=edges.end();++i)
195       outfile<<((Edge)(*i)).p1().x()<<"␣"<<((Edge)(*i)).p1().y()
196           <<"␣"<<((Edge)(*i)).p2().x()<<"␣"<<((Edge)(*i)).p2().y()<<"␣dl\n";
197     outfile<<"showpage\n";
198 }
199 
200 //The file v_filename containes the coordinates of the vertices. The
201 //file s_filename containes the numbers of the vertices for each
202 //shape of the triangulation. Note that since this routine was writen
203 //with the intention to provide easy exoprt-possibility to the matlab
204 //environment the first vertice has the index 1 and not the index 0!
205 void
```

```
206 write_matlab_tr3d(const char* v_filename,const char* s_filename,
207                   const Triangulation3D& tr_to_write)
208 {
209   std::ofstream v_outfile(v_filename);
210   std::ofstream s_outfile(s_filename);
211   if(!v_outfile || !s_outfile)
212   {
213     std::cerr<<"Could␣not␣open␣one␣of␣the␣files␣"<<v_filename<<"␣or␣"<<s_filename;
214     return;
215   }
216
217   std::vector<Point3D*> vertices=std::vector<Point3D*>();
218   std::map<Point3D,int> vertices_index=std::map<Point3D,int>();
219
220   //Gather all the information from the Triangulation3D specially
221   //assigning and determining the index relations between vertices
222   //and edges
223   for(Triangulation3D::const_iterator i=tr_to_write.begin();
224       i!=tr_to_write.end();++i)
225   {
226     static std::pair<std::map<Point3D,int>::iterator,bool> insert_info_v;
227     static std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
228     edges_of_triangle=(*i)->get_edges();
229
230     //if a new vertice is found - insert with index value
231     insert_info_v=vertices_index.insert(
232         std::pair<Point3D,int>(edges_of_triangle[0]->p1(),
233                                vertices_index.size()+1));
234     if(insert_info_v.second)
235       vertices.push_back((Point3D*)&(insert_info_v.first->first));
236     insert_info_v=vertices_index.insert(
237         std::pair<Point3D,int>(edges_of_triangle[1]->p1(),
238                                vertices_index.size()+1));
239     if(insert_info_v.second)
240       vertices.push_back((Point3D*)&(insert_info_v.first->first));
241     insert_info_v=vertices_index.insert(
242         std::pair<Point3D,int>(edges_of_triangle[2]->p1(),
243                                vertices_index.size()+1));
244     if(insert_info_v.second)
245       vertices.push_back((Point3D*)&(insert_info_v.first->first));
246   }
247   //write all the info to the specified v_file.
248   for(int i=0;i<vertices.size();++i)
249     v_outfile<<vertices[i]->x()<<"␣"<<vertices[i]->y()<<"␣"
250              <<vertices[i]->z()<<"\n";
251
252   //write the shape info to the specified s_file
253   for(Triangulation3D::const_iterator i=tr_to_write.begin();
254       i!=tr_to_write.end();++i)
255   {
256     static std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
257     edges_of_triangle=(*i)->get_edges();
258
259     for(int j=0;j<edges_of_triangle.size();++j)
260       s_outfile<<vertices_index[edges_of_triangle[j]->p1()]<<"␣";
261     s_outfile<<"\n";
262   }
263 }
264
265 void
266 write_matlab_visualisation_data_tr3d(const char* x_filename,
267                                      const char* y_filename,
268                                      const char* z_filename,
```

```
269                                     const Triangulation3D& mesh)
270 {
271   std::ofstream x_file(x_filename);
272   std::ofstream y_file(y_filename);
273   std::ofstream z_file(z_filename);
274   if(!x_file || !y_file || !z_file)
275   {
276     std::cerr<<"Could␣not␣open␣one␣of␣the␣files␣"<<x_filename<<",␣"
277              <<y_filename<<"␣or␣"<<z_filename;
278     return;
279   }
280   if(mesh[0]->what_are_you()==TRIANGLE)
281     for(int i=0;i<mesh.size();++i)
282     {
283       std::vector<Point3D> vertices=mesh[i]->get_vertices();
284       x_file<<vertices[0].x()<<"␣"<<vertices[1].x()<<"␣"
285             <<vertices[2].x()<<"\n";
286       y_file<<vertices[0].y()<<"␣"<<vertices[1].y()<<"␣"
287             <<vertices[2].y()<<"\n";
288       z_file<<vertices[0].z()<<"␣"<<vertices[1].z()<<"␣"
289             <<vertices[2].z()<<"\n";
290     }
291   else
292     for(int i=0;i<mesh.size();++i)
293     {
294       std::vector<Point3D> vertices=mesh[i]->get_vertices();
295       x_file<<vertices[0].x()<<"␣"<<vertices[1].x()<<"␣"<<vertices[2].x()<<
296           "␣"<<vertices[3].x()<<"\n";
297       y_file<<vertices[0].y()<<"␣"<<vertices[1].y()<<"␣"<<vertices[2].y()<<
298           "␣"<<vertices[3].y()<<"\n";
299       z_file<<vertices[0].z()<<"␣"<<vertices[1].z()<<"␣"<<vertices[2].z()<<
300           "␣"<<vertices[3].z()<<"\n";
301     }
302 }
303
304 #ifdef HAVE_HLIB
305 //Definition of a HLib compatible structure to represent triangles
306 typedef struct _hlib_triangle_
307 {
308   int vertices[3];
309   int edges[3];
310 }hlib_triangle;
311
312 typedef struct _hlib_edge_
313 {
314   int vertices[2];
315 }hlib_edge;
316 //create a mesh in the hlib format from an existing Triangulation3D object.
317 //Note! It is assumed, that the triangulation3d contains only triangles
318 pbemgrid3d triangulation3d_to_bemgrid3d(const Triangulation3D& tr_to_create)
319 {
320   std::vector<Point3D> vertices=std::vector<Point3D>();
321   std::vector<Edge> edges=std::vector<Edge>();
322   std::vector<hlib_edge*> h_edges=std::vector<hlib_edge*>();
323   std::vector<hlib_triangle*> h_triangles=std::vector<hlib_triangle*>();
324
325   //Gather all the information from the Triangulation3D specially
326   //assigning and determining the index relations between vertices,
327   //edges and triangles
328   //FIRST IDENTIFY ALL VERTICES
329   for(int i=0;i<tr_to_create.size();++i)
330   {
331     bool found=false;
```

```
332
333     hlib_triangle* actual_triangle=new hlib_triangle;
334     std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
335     edges_of_triangle=tr_to_create[i]->get_edges();
336     std::vector<Point3D> vertices_of_triangle=std::vector<Point3D>();
337     vertices_of_triangle=tr_to_create[i]->get_vertices();
338
339     int index=0;
340     for(int j=0;j<vertices.size();j++)
341       if(vertices_of_triangle[0]==vertices[j])
342       {
343         found=true;
344         index=j;
345         break;
346       }
347     if(!found)
348     {
349       actual_triangle->vertices[0]=vertices.size();
350       vertices.push_back(vertices_of_triangle[0]);
351     }
352     else
353       actual_triangle->vertices[0]=index;
354
355     found=false;
356     for(int j=0;j<vertices.size();j++)
357       if(vertices_of_triangle[1]==vertices[j])
358     {
359       found=true;
360       index=j;
361       break;
362     }
363     if(!found)
364     {
365       actual_triangle->vertices[1]=vertices.size();
366       vertices.push_back(vertices_of_triangle[1]);
367     }
368     else
369       actual_triangle->vertices[1]=index;
370
371     found=false;
372     for(int j=0;j<vertices.size();j++)
373       if(vertices_of_triangle[2]==vertices[j])
374     {
375       found=true;
376       index=j;
377       break;
378     }
379     if(!found)
380     {
381       actual_triangle->vertices[2]=vertices.size();
382       vertices.push_back(vertices_of_triangle[2]);
383     }
384     else
385       actual_triangle->vertices[2]=index;
386
387     h_triangles.push_back(actual_triangle);
388   }
389   std::cout<<"Triangulation-Export:␣Vertices␣idetified!\n";
390
391   //NOW IDENTIFY THE EDGES
392   for(int i=0;i<tr_to_create.size();++i)
393   {
394     int index=0;
```

```
395     hlib_triangle* actual_triangle=h_triangles[i];
396     std::vector<ShapeEdge*> edges_of_triangle=std::vector<ShapeEdge*>();
397     edges_of_triangle=tr_to_create[i]->get_edges();
398     std::vector<Point3D> vertices_of_triangle=std::vector<Point3D>();
399     vertices_of_triangle=tr_to_create[i]->get_vertices();
400
401     bool found=false;
402     for(int j=0;j<edges.size();++j)
403       if(*(edges_of_triangle[0])==edges[j])
404       {
405         found=true;
406         index=j;
407         break;
408       }
409     if(!found)
410     {
411       hlib_edge* actual_edge=new hlib_edge;
412       int index2=0;
413       for(int k=0;k<vertices.size();++k)
414         if(vertices[k]==(*(edges_of_triangle[0])).p1())
415         {
416           index2=k;
417           break;
418         }
419       actual_edge->vertices[0]=index2;
420       for(int k=0;k<vertices.size();++k)
421         if(vertices[k]==(*(edges_of_triangle[0])).p2())
422         {
423           index2=k;
424           break;
425         }
426       actual_edge->vertices[1]=index2;
427       actual_triangle->edges[0]=edges.size();
428       edges.push_back(*(edges_of_triangle[0]));
429       h_edges.push_back(actual_edge);
430     }
431     else
432       actual_triangle->edges[0]=index;
433
434     found=false;
435     for(int j=0;j<edges.size();++j)
436       if(*(edges_of_triangle[1])==edges[j])
437       {
438         found=true;
439         break;
440       }
441       if(!found)
442       {
443         hlib_edge* actual_edge=new hlib_edge;
444         int index2=0;
445         for(int k=0;k<vertices.size();++k)
446           if(vertices[k]==(*(edges_of_triangle[1])).p1())
447         {
448           index2=k;
449           break;
450         }
451         actual_edge->vertices[0]=index2;
452         for(int k=0;k<vertices.size();++k)
453           if(vertices[k]==(*(edges_of_triangle[1])).p2())
454         {
455           index2=k;
456           break;
457         }
```

```
458         actual_edge->vertices[1]=index2;
459         actual_triangle->edges[1]=edges.size();
460         edges.push_back(*(edges_of_triangle[1]));
461         h_edges.push_back(actual_edge);
462       }
463       else
464         actual_triangle->edges[1]=index;
465
466     found=false;
467     for(int j=0;j<edges.size();++j)
468       if(*(edges_of_triangle[2])==edges[j])
469     {
470       found=true;
471       break;
472     }
473     if(!found)
474     {
475       hlib_edge* actual_edge=new hlib_edge;
476       int index2=0;
477       for(int k=0;k<vertices.size();++k)
478         if(vertices[k]==(*(edges_of_triangle[2])).p1())
479       {
480         index2=k;
481         break;
482       }
483       actual_edge->vertices[0]=index2;
484       for(int k=0;k<vertices.size();++k)
485         if(vertices[k]==(*(edges_of_triangle[2])).p2())
486       {
487         index2=k;
488         break;
489       }
490       actual_edge->vertices[1]=index2;
491       actual_triangle->edges[2]=edges.size();
492       edges.push_back(*(edges_of_triangle[2]));
493       h_edges.push_back(actual_edge);
494     }
495     else
496       actual_triangle->edges[2]=index;
497   }
498   std::cout<<"Triangulation-Export:␣Edges␣idetified!\n";
499   //create the bemgrid3d with correct number of triangles, vertices
500   //and edges
501   pbemgrid3d newbemgrid=new_bemgrid3d(vertices.size(),edges.size(),
502                                       tr_to_create.size());
503   std::cout<<"Triangulation-Export:␣Empty␣bemgrid3d␣allocated!\n";
504
505
506   //write the vertices coordinates
507   for(int i=0;i<vertices.size();++i)
508   {
509     newbemgrid->x[i][0]=vertices[i].x();
510     newbemgrid->x[i][1]=vertices[i].y();
511     newbemgrid->x[i][2]=vertices[i].z();
512   }
513   std::cout<<"Triangulation-Export:␣Vertices␣writen␣to␣bemgrid3d!\n";
514
515   //write the number of the edges endpoints
516   for(int i=0;i<h_edges.size();++i)
517   {
518     newbemgrid->e[i][0]=h_edges[i]->vertices[0];
519     newbemgrid->e[i][1]=h_edges[i]->vertices[1];
520     delete h_edges[i];
```

```
521   }
522   std::cout<<"Triangulation-Export:␣Edges␣writen␣to␣bemgrid3d!\n";
523 
524   //write the number of the triangles vertices and edges
525   for(int i=0;i<h_triangles.size();++i)
526   {
527     newbemgrid->s[i][0]=h_triangles[i]->edges[0];
528     newbemgrid->s[i][1]=h_triangles[i]->edges[1];
529     newbemgrid->s[i][2]=h_triangles[i]->edges[2];
530     newbemgrid->t[i][0]=h_triangles[i]->vertices[0];
531     newbemgrid->t[i][1]=h_triangles[i]->vertices[1];
532     newbemgrid->t[i][2]=h_triangles[i]->vertices[2];
533     delete h_triangles[i];
534   }
535   std::cout<<"Triangulation-Export:␣Triangles␣writen␣to␣bemgrid3d!\n";
536 
537   //finally compute the outer normal vectors and the gram determinants
538   //using the allready implemented functions from the hlib
539   prepare_bemgrid3d(newbemgrid);
540 
541   return newbemgrid;
542 }
543 #endif
544 
545 Triangulation3D create_graded_mesh(const char* koords,const char* elements)
546 {
547   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
548   std::ifstream infile1(koords);
549   std::ifstream infile2(elements);
550   std::vector<Point3D> pts=std::vector<Point3D>();
551   while(!infile1.eof())
552   {
553     double koord=0.;
554     infile1>>koord;
555     Point3D p=Point3D();
556     p.set_x(koord);
557     infile1>>koord;
558     p.set_y(koord);
559     pts.push_back(p);
560     char c=infile1.get();
561     while(isspace(c) && !infile1.eof())
562       c=infile1.get();
563     if(!infile1.eof())
564       infile1.putback(c);
565 
566   }
567   infile1.close();
568   int help=0;
569   while(infile2>>help && !infile2.eof())
570   {
571     int i=help;
572     int j=0;
573     int k=0;
574     int l=0;
575     infile2>>j;
576     infile2>>k;
577     infile2>>l;
578     --i;--j;--k;--l;
579     Rectangle* R=new Rectangle(pts[i],pts[j],pts[k],pts[l]);
580     init_triangulation.push_back(R);
581   }
582   infile2.close();
583   Triangulation3D triangulation=Triangulation3D(init_triangulation);
```

```
584   return triangulation;
585 }
586
587 Triangulation3D create_graded_cube_from_square(const char* koords,
588                                                const char* elements)
589 {
590   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
591   std::ifstream infile1(koords);
592   std::ifstream infile2(elements);
593   std::vector<Point3D> pts=std::vector<Point3D>();
594   while(!infile1.eof())
595   {
596     double koord=0.;
597     infile1>>koord;
598     Point3D p=Point3D();
599     p.set_x(koord);
600     infile1>>koord;
601     p.set_y(koord);
602     pts.push_back(p);
603     char c=infile1.get();
604     while(isspace(c) && !infile1.eof())
605       c=infile1.get();
606     if(!infile1.eof())
607       infile1.putback(c);
608
609   }
610   infile1.close();
611   int help=0;
612   while(infile2>>help && !infile2.eof())
613   {
614     int i=help;
615     int j=0;
616     int k=0;
617     int l=0;
618     infile2>>j;
619     infile2>>k;
620     infile2>>l;
621     --i;--j;--k;--l;
622     Rectangle* R=new Rectangle(pts[i],pts[j],pts[k],pts[l]);
623     init_triangulation.push_back(R);
624   }
625   infile2.close();
626   //Up to this point the init-triangulation consist of the square (0,1)^2.
627   //Now some basic geometric manipulation and copying of Rectangles is done
628   //to obtain the cube (-1,1)^3
629   std::vector<Shape*> more_init_triag=std::vector<Shape*>();
630   for(int i=0;i<init_triangulation.size();++i)
631   {
632     Matrix x_rot=Matrix(3);
633     x_rot.set(1,1,1);
634     x_rot.set(1,2,0);
635     x_rot.set(1,3,0);
636     x_rot.set(2,1,0);
637     x_rot.set(2,2,0);
638     x_rot.set(2,3,-1);
639     x_rot.set(3,1,0);
640     x_rot.set(3,2,1);
641     x_rot.set(3,3,0);
642     Matrix y_rot=Matrix(3);
643     y_rot.set(1,1,0);
644     y_rot.set(1,2,0);
645     y_rot.set(1,3,1);
646     y_rot.set(2,1,0);
```

```
647       y_rot.set(2,2,1);
648       y_rot.set(2,3,0);
649       y_rot.set(3,1,1);
650       y_rot.set(3,2,0);
651       y_rot.set(3,3,0);
652
653       std::vector<Point3D> v=init_triangulation[i]->get_vertices();
654       Vector e=Vector(1,1,1);
655       Point3D a=MVM(y_rot,v[0]);
656       Point3D b=MVM(y_rot,v[1]);
657       Point3D c=MVM(y_rot,v[2]);
658       Point3D d=MVM(y_rot,v[3]);
659       Vector ex=Vector(1.,0.,0.);
660       Vector ey=Vector(0.,1.,0.);
661       Vector ez=Vector(0.,0.,1.);
662       Rectangle* left=new Rectangle(a*2-e,b*2-e,c*2-e,d*2-e);
663       Rectangle* right=new Rectangle((a+ex)*2-e,(b+ex)*2-e,(c+ex)*2-e,(d+ex)*2-e);
664       a=MVM(x_rot,v[0]);
665       b=MVM(x_rot,v[1]);
666       c=MVM(x_rot,v[2]);
667       d=MVM(x_rot,v[3]);
668       Rectangle* near=new Rectangle(a*2-e,b*2-e,c*2-e,d*2-e);
669       Rectangle* far=new Rectangle((a+ey)*2-e,(b+ey)*2-e,(c+ey)*2-e,(d+ey)*2-e);
670       Rectangle* top=new Rectangle((v[0]+ez)*2-e,(v[1]+ez)*2-e,
671                                    (v[2]+ez)*2-e,(v[3]+ez)*2-e);
672       Rectangle* bottom=new Rectangle(v[0]*2-e,v[1]*2-e,v[2]*2-e,v[3]*2-e);
673
674       more_init_triag.push_back(bottom);
675       more_init_triag.push_back(top);
676       more_init_triag.push_back(left);
677       more_init_triag.push_back(right);
678       more_init_triag.push_back(near);
679       more_init_triag.push_back(far);
680     }
681     //Free the Rectangles in init_triag
682     for(int i=0;i<init_triangulation.size();++i)
683       free(init_triangulation[i]);
684
685     Triangulation3D triangulation=Triangulation3D(more_init_triag);
686     return triangulation;
687   }
688
689   Triangulation3D create_mesh_from_visualisationdata(const char* xdat,
690                                                      const char* ydat,
691                                                      const char* zdat,
692                                                      bool allow_hanging,
693                                                      bool consider_neighbours)
694   {
695     std::vector<Shape*> init_triangulation=std::vector<Shape*>();
696     std::ifstream x(xdat);
697     std::ifstream y(ydat);
698     std::ifstream z(zdat);
699
700     while(!x.eof())
701     {
702       double koord=0.;
703       Point3D a=Point3D();
704       Point3D b=Point3D();
705       Point3D c=Point3D();
706       Point3D d=Point3D();
707       x>>koord;
708       a.set_x(koord);
709       y>>koord;
```

```
710       a.set_y(koord);
711       z>>koord;
712       a.set_z(koord);
713
714       x>>koord;
715       b.set_x(koord);
716       y>>koord;
717       b.set_y(koord);
718       z>>koord;
719       b.set_z(koord);
720
721       x>>koord;
722       c.set_x(koord);
723       y>>koord;
724       c.set_y(koord);
725       z>>koord;
726       c.set_z(koord);
727
728       x>>koord;
729       d.set_x(koord);
730       y>>koord;
731       d.set_y(koord);
732       z>>koord;
733       d.set_z(koord);
734
735       Rectangle* R=new Rectangle(a,b,c,d,allow_hanging);
736
737       init_triangulation.push_back(R);
738
739       char tmp=x.get();
740       while(isspace(tmp) && !x.eof())
741         tmp=x.get();
742       if(!x.eof())
743         x.putback(tmp);
744     }
745
746     if(consider_neighbours)
747       for(int i=0;i<init_triangulation.size();++i)
748         for(int j=i+1;j<init_triangulation.size();++j)
749         {
750           init_triangulation[i]->add_neighbour(init_triangulation[j]);
751           init_triangulation[j]->add_neighbour(init_triangulation[i]);
752         }
753
754     Triangulation3D T=Triangulation3D(init_triangulation);
755     return T;
756   }
757
758   }
```

A.3.7 usual_meshes.h

```
1  //*************************************************************************//
2  //File:                    usual_meshes.h                                  //
3  //This file specifies three routines to create meshes, which can be used as  //
4  //initial mesh for computations. Concrete it specifies functions to create   //
5  //a box (-1,1)^3, a L-Screen and Fichera's cube.                             //
6  //*************************************************************************//
7  //Author: Samuel Ferraz-Leite                                               //
8  //Date: 07.2007                                                             //
9  //Version 1.0 stable                                                        //
10 //*************************************************************************//
11 #ifndef USUAL_MESHES_H
12 #define USUAL_MESHES_H
```

```
13
14 //Includes of the FLBEM-modules
15 #include "triangulation.h"
16 #include "triangulationio.h"
17
18 namespace triangulation{
19
20 //creates a box mesh of the dirichlet-type.
21 Triangulation3D create_box_mesh(shape_type shape_to_use, bool allow_hanging,
22                                 bool consider_neighbourhoods=true);
23
24 //create an L-shaped 2D-border of dirichlet-type.
25 Triangulation3D create_L2D_mesh(shape_type shape_to_use, bool allow_hanging,
26                                bool consider_neighbourhoods=true);
27
28 //crate a mesh on Fichera's cube of dirichlet-type
29 Triangulation3D create_fichera_mesh(shape_type shape_to_use, bool allow_hanging,
30                                    bool consider_neighbourhoods=true);
31 }
32 #endif
```

A.3.8 usual_meshes.cpp

```
1  //************************************************************************//
2  //File:                    usual_meshes.cpp                                 //
3  //This file implements the functions deklared in usual_meshes.h             //
4  //************************************************************************//
5  //Author: Samuel Ferraz-Leite                                               //
6  //Date: 07.2007                                                             //
7  //Version 1.0 stable                                                        //
8  //************************************************************************//
9
10 //Include of the FLBEM-modules
11 #include "triangulation.h"
12 #include "triangulationio.h"
13 #include "usual_meshes.h"
14
15 //Includes of the standard-libraries
16 #include <list>
17 #include <vector>
18 #include <iostream>
19
20
21 namespace triangulation{
22
23 Triangulation3D create_box_mesh(shape_type shape_to_use, bool allow_hanging,
24                                 bool consider_neighbourhood)
25 {
26   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
27   if (shape_to_use==TRIANGLE)
28   {
29     Triangle* T1=new Triangle(Point3D(-1,-1,-1),Point3D(1,-1,-1),
30                               Point3D(-1,-1,1),allow_hanging);
31     Triangle* T2=new Triangle(Point3D(-1,-1,1),Point3D(1,-1,1),
32                               Point3D(1,-1,-1),allow_hanging);
33     Triangle* T3=new Triangle(Point3D(1,-1,-1),Point3D(1,-1,1),
34                               Point3D(1,1,-1),allow_hanging);
35     Triangle* T4=new Triangle(Point3D(1,-1,1),Point3D(1,1,1),
36                               Point3D(1,1,-1),allow_hanging);
37     Triangle* T5=new Triangle(Point3D(1,1,1),Point3D(1,1,-1),
38                               Point3D(-1,1,-1),allow_hanging);
39     Triangle* T6=new Triangle(Point3D(1,1,1),Point3D(-1,1,1),
40                               Point3D(-1,1,-1),allow_hanging);
41     Triangle* T7=new Triangle(Point3D(-1,-1,1),Point3D(-1,1,1),
```

```
42                                Point3D(-1,1,-1),allow_hanging);
43      Triangle* T8=new Triangle(Point3D(-1,-1,1),Point3D(-1,-1,-1),
44                                Point3D(-1,1,-1),allow_hanging);
45      Triangle* T9=new Triangle(Point3D(1,1,-1),Point3D(-1,1,-1),
46                                Point3D(1,-1,-1),allow_hanging);
47      Triangle* T10=new Triangle(Point3D(1,-1,-1),Point3D(-1,-1,-1),
48                                 Point3D(-1,1,-1),allow_hanging);
49      Triangle* T11=new Triangle(Point3D(1,1,1),Point3D(-1,1,1),
50                                 Point3D(1,-1,1),allow_hanging);
51      Triangle* T12=new Triangle(Point3D(1,-1,1),Point3D(-1,-1,1),
52                                 Point3D(-1,1,1),allow_hanging);
53
54
55      init_triangulation.push_back(T1);
56      init_triangulation.push_back(T2);
57      init_triangulation.push_back(T3);
58      init_triangulation.push_back(T4);
59      init_triangulation.push_back(T5);
60      init_triangulation.push_back(T6);
61      init_triangulation.push_back(T7);
62      init_triangulation.push_back(T8);
63      init_triangulation.push_back(T9);
64      init_triangulation.push_back(T10);
65      init_triangulation.push_back(T11);
66      init_triangulation.push_back(T12);
67
68      if(consider_neighbourhood)
69      {
70        for(std::vector<Shape*>::iterator i=init_triangulation.begin();
71            i!=init_triangulation.end();++i)
72        {
73          std::vector<Shape*>::iterator j=i;
74          ++j;
75          while(j!=init_triangulation.end())
76          {
77            (*i)->add_neighbour(*j);
78            (*j)->add_neighbour(*i);
79            ++j;
80          }
81        }
82      }
83    }
84    else
85    {
86      Rectangle* R1=new
87          Rectangle(Point3D(-1,-1,-1),Point3D(-1,1,-1),Point3D(1,1,-1),
88                    Point3D(1,-1,-1),allow_hanging);
89      Rectangle* R2=new
90          Rectangle(Point3D(-1,-1,-1),Point3D(-1,-1,1),Point3D(-1,1,1),
91                    Point3D(-1,1,-1),allow_hanging);
92      Rectangle* R3=new
93          Rectangle(Point3D(-1,1,-1),Point3D(-1,1,1),Point3D(1,1,1),
94                    Point3D(1,1,-1),allow_hanging);
95      Rectangle* R4=new
96          Rectangle(Point3D(1,1,-1),Point3D(1,1,1),Point3D(1,-1,1),
97                    Point3D(1,-1,-1),allow_hanging);
98      Rectangle* R5=new
99          Rectangle(Point3D(1,-1,-1),Point3D(1,-1,1),Point3D(-1,-1,1),
100                   Point3D(-1,-1,-1),allow_hanging);
101     Rectangle* R6=new
102         Rectangle(Point3D(-1,-1,1),Point3D(1,-1,1),Point3D(1,1,1),
103                   Point3D(-1,1,1),allow_hanging);
104
```

```
105     init_triangulation.push_back(R1);
106     init_triangulation.push_back(R2);
107     init_triangulation.push_back(R3);
108     init_triangulation.push_back(R4);
109     init_triangulation.push_back(R5);
110     init_triangulation.push_back(R6);
111
112     if(consider_neighbourhood)
113     {
114       for(std::vector<Shape*>::iterator i=init_triangulation.begin();
115           i!=init_triangulation.end();++i)
116       {
117         std::vector<Shape*>::iterator j=i;
118         ++j;
119         while(j!=init_triangulation.end())
120         {
121           (*i)->add_neighbour(*j);
122           (*j)->add_neighbour(*i);
123           ++j;
124         }
125       }
126     }
127   }
128
129   Triangulation3D triangulation=Triangulation3D(init_triangulation,
130                                                 consider_neighbourhood);
131   return triangulation;
132 }
133
134 Triangulation3D create_L2D_mesh(shape_type shape_to_use, bool allow_hanging,
135                                 bool consider_neighbourhoods)
136 {
137   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
138   if(shape_to_use==TRIANGLE)
139   {
140     Triangle* T1=new Triangle(Point3D(0,2,0),Point3D(1,2,0),
141                               Point3D(0,1,0),allow_hanging);
142     Triangle* T2=new Triangle(Point3D(0,1,0),Point3D(1,1,0),
143                               Point3D(1,2,0),allow_hanging);
144     Triangle* T3=new Triangle(Point3D(0,1,0),Point3D(1,1,0),
145                               Point3D(0,0,0),allow_hanging);
146     Triangle* T4=new Triangle(Point3D(0,0,0),Point3D(1,0,0),
147                               Point3D(1,1,0),allow_hanging);
148     Triangle* T5=new Triangle(Point3D(1,1,0),Point3D(1,0,0),
149                               Point3D(2,1,0),allow_hanging);
150     Triangle* T6=new Triangle(Point3D(1,0,0),Point3D(2,0,0),
151                               Point3D(2,1,0),allow_hanging);
152
153     init_triangulation.push_back(T1);
154     init_triangulation.push_back(T2);
155     init_triangulation.push_back(T3);
156     init_triangulation.push_back(T4);
157     init_triangulation.push_back(T5);
158     init_triangulation.push_back(T6);
159
160     if(consider_neighbourhoods)
161     {
162       for(std::vector<Shape*>::iterator i=init_triangulation.begin();
163           i!=init_triangulation.end();++i)
164       {
165         std::vector<Shape*>::iterator j=i;
166         ++j;
167         while(j!=init_triangulation.end())
```

```
168         {
169           (*i)->add_neighbour(*j);
170           (*j)->add_neighbour(*i);
171           ++j;
172         }
173       }
174     }
175   }
176   else
177   {
178     Rectangle* R1=new
179         Rectangle(Point3D(0,0,0),Point3D(1,0,0),Point3D(1,1,0),
180                   Point3D(0,1,0),allow_hanging);
181     Rectangle* R2=new
182         Rectangle(Point3D(0,1,0),Point3D(1,1,0),Point3D(1,2,0),
183                   Point3D(0,2,0),allow_hanging);
184     Rectangle* R3=new
185         Rectangle(Point3D(1,0,0),Point3D(2,0,0),Point3D(2,1,0),
186                   Point3D(1,1,0),allow_hanging);
187
188     init_triangulation.push_back(R1);
189     init_triangulation.push_back(R2);
190     init_triangulation.push_back(R3);
191
192     if(consider_neighbourhoods)
193     {
194       for(std::vector<Shape*>::iterator i=init_triangulation.begin();
195           i!=init_triangulation.end();++i)
196       {
197         std::vector<Shape*>::iterator j=i;
198         ++j;
199         while(j!=init_triangulation.end())
200         {
201           (*i)->add_neighbour(*j);
202           (*j)->add_neighbour(*i);
203           ++j;
204         }
205       }
206     }
207   }
208
209   Triangulation3D triangulation=Triangulation3D(init_triangulation,
210                                                 consider_neighbourhoods);
211   return triangulation;
212 }
213
214 Triangulation3D create_fichera_mesh(shape_type shape_to_use,
215                                     bool allow_hanging,
216                                     bool consider_neighbourhoods)
217 {
218   std::vector<Shape*> init_triangulation=std::vector<Shape*>();
219     //lower side
220     Rectangle* R1=new
221         Rectangle(Point3D(-1,-1,-1),Point3D(0,-1,-1),Point3D(0,0,-1),
222                   Point3D(-1,0,-1),allow_hanging);
223     Rectangle* R2=new
224         Rectangle(Point3D(-1,0,-1),Point3D(0,0,-1),Point3D(0,1,-1),
225                   Point3D(-1,1,-1),allow_hanging);
226     Rectangle* R3=new
227         Rectangle(Point3D(0,-1,-1),Point3D(1,-1,-1),Point3D(1,0,-1),
228                   Point3D(0,0,-1),allow_hanging);
229     Rectangle* R4=new
230         Rectangle(Point3D(0,0,-1),Point3D(1,0,-1),Point3D(1,1,-1),
```

```
231                 Point3D(0,1,-1),allow_hanging);
232
233     //upper side
234     Rectangle* R5=new
235         Rectangle(Point3D(-1,-1,1),Point3D(0,-1,1),Point3D(0,0,1),
236                 Point3D(-1,0,1),allow_hanging);
237     Rectangle* R6=new
238         Rectangle(Point3D(-1,0,1),Point3D(0,0,1),Point3D(0,1,1),
239                 Point3D(-1,1,1),allow_hanging);
240     Rectangle* R7=new
241         Rectangle(Point3D(0,0,1),Point3D(1,0,1),Point3D(1,1,1),
242                 Point3D(0,1,1),allow_hanging);
243
244     //left side
245     Rectangle* R8=new
246         Rectangle(Point3D(-1,-1,-1),Point3D(-1,0,-1),Point3D(-1,0,0),
247                 Point3D(-1,-1,0),allow_hanging);
248     Rectangle* R9=new
249         Rectangle(Point3D(-1,0,-1),Point3D(-1,1,-1),Point3D(-1,1,0),
250                 Point3D(-1,0,0),allow_hanging);
251     Rectangle* R10=new
252         Rectangle(Point3D(-1,-1,0),Point3D(-1,0,0),Point3D(-1,0,1),
253                 Point3D(-1,-1,1),allow_hanging);
254     Rectangle* R11=new
255         Rectangle(Point3D(-1,0,0),Point3D(-1,1,0),Point3D(-1,1,1),
256                 Point3D(-1,0,1),allow_hanging);
257
258     //back side
259     Rectangle* R12=new
260         Rectangle(Point3D(-1,1,-1),Point3D(0,1,-1),Point3D(0,1,0),
261                 Point3D(-1,1,0),allow_hanging);
262     Rectangle* R13=new
263         Rectangle(Point3D(0,1,-1),Point3D(1,1,-1),Point3D(1,1,0),
264                 Point3D(0,1,0),allow_hanging);
265     Rectangle* R14=new
266         Rectangle(Point3D(-1,1,0),Point3D(0,1,0),Point3D(0,1,1),
267                 Point3D(-1,1,1),allow_hanging);
268     Rectangle* R15=new
269         Rectangle(Point3D(0,1,0),Point3D(1,1,0),Point3D(1,1,1),
270                 Point3D(0,1,1),allow_hanging);
271
272     //front side
273     Rectangle* R16=new
274         Rectangle(Point3D(-1,-1,-1),Point3D(0,-1,-1),Point3D(0,-1,0),
275                 Point3D(-1,-1,0),allow_hanging);
276     Rectangle* R17=new
277         Rectangle(Point3D(0,-1,-1),Point3D(1,-1,-1),Point3D(1,-1,0),
278                 Point3D(0,-1,0),allow_hanging);
279     Rectangle* R18=new
280         Rectangle(Point3D(-1,-1,0),Point3D(0,-1,0),Point3D(0,-1,1),
281                 Point3D(-1,-1,1),allow_hanging);
282
283     //right side
284     Rectangle* R19=new
285         Rectangle(Point3D(1,-1,-1),Point3D(1,0,-1),Point3D(1,0,0),
286                 Point3D(1,-1,0),allow_hanging);
287     Rectangle* R20=new
288         Rectangle(Point3D(1,0,-1),Point3D(1,1,-1),Point3D(1,1,0),
289                 Point3D(1,0,0),allow_hanging);
290     Rectangle* R21=new
291         Rectangle(Point3D(1,0,0),Point3D(1,1,0),Point3D(1,1,1),
292                 Point3D(1,0,1),allow_hanging);
293
```

```
294     //still missing 3 small rectangles
295     Rectangle* R22=new
296         Rectangle(Point3D(0,-1,0),Point3D(0,0,0),Point3D(0,0,1),
297                   Point3D(0,-1,1),allow_hanging);
298     Rectangle* R23=new
299         Rectangle(Point3D(0,0,0),Point3D(1,0,0),Point3D(1,0,1),
300                   Point3D(0,0,1),allow_hanging);
301     Rectangle* R24=new
302         Rectangle(Point3D(0,-1,0),Point3D(1,-1,0),Point3D(1,0,0),
303                   Point3D(0,0,0),allow_hanging);
304
305     init_triangulation.push_back(R1);
306     init_triangulation.push_back(R2);
307     init_triangulation.push_back(R3);
308     init_triangulation.push_back(R4);
309     init_triangulation.push_back(R5);
310     init_triangulation.push_back(R6);
311     init_triangulation.push_back(R7);
312     init_triangulation.push_back(R8);
313     init_triangulation.push_back(R9);
314     init_triangulation.push_back(R10);
315     init_triangulation.push_back(R11);
316     init_triangulation.push_back(R12);
317     init_triangulation.push_back(R13);
318     init_triangulation.push_back(R14);
319     init_triangulation.push_back(R15);
320     init_triangulation.push_back(R16);
321     init_triangulation.push_back(R17);
322     init_triangulation.push_back(R18);
323     init_triangulation.push_back(R19);
324     init_triangulation.push_back(R20);
325     init_triangulation.push_back(R21);
326     init_triangulation.push_back(R22);
327     init_triangulation.push_back(R23);
328     init_triangulation.push_back(R24);
329
330     if(consider_neighbourhoods)
331     {
332       for(std::vector<Shape*>::iterator i=init_triangulation.begin();
333           i!=init_triangulation.end();++i)
334       {
335         std::vector<Shape*>::iterator j=i;
336         ++j;
337         while(j!=init_triangulation.end())
338         {
339           (*i)->add_neighbour(*j);
340           (*j)->add_neighbour(*i);
341           ++j;
342         }
343       }
344     }
345     // }
346
347   Triangulation3D triangulation=Triangulation3D(init_triangulation,
348                                                 consider_neighbourhoods);
349   return triangulation;
350 }
351
352 }
```

A.4 Integraloperatoren

A.4.1 maiprogs.h

```
//*************************************************************************//
//File:                    maiprogs.h                                        //
//This very short file does nothing other than to declare the function      //
//lapinteg3 from the maiprogs library, so that it can be used by            //
//integral_operators module to analytically compute the entries to the      //
//stiffness matrix.                                                         //
//*************************************************************************//
//Author: Samuel Ferraz-Leite                                               //
//Date: 07.2007                                                             //
//Version 1.0 stable                                                        //
//*************************************************************************//
#ifndef BEM_MAIPROGS
#define BEM_MAIPROGS

void liblap3_MP_lapinteg3(double* bx,double* dx1,double* dx2,double* nx,int* tx,
                          double* by,double* dy1,double* dy2,double* ny,int* ty,
                          int* pvx,int* pvy,int* pkx,int* pky,int* pwx,int*pwy,
                          double* vklmn,double* kklmn,double* wklmn);
#endif
```

A.4.2 quadrature.h

```
//*************************************************************************//
//File:                    quadrature.h                                      //
//This file declares the antiderivatives necessary to compute the entries   //
//of the stiffness matrix for BEM. The name of the functions and variables  //
//follow the names provided in the paper of M. Maischaik, wehere these      //
//antiderivatives are taken from. For more details I suggest you read       //
//the paper.                                                                //
//*************************************************************************//
//Author: Samuel Ferraz-Leite                                               //
//Date: 07.2007                                                             //
//Version 0.8 mot all features included                                     //
//*************************************************************************//
#ifndef BEMLIB_QUADRATURE_H
#define BEMLIB_QUADRATURE_H

//Include the standard-libraries
#include<cmath>
#include<vector>
#include<list>
#include<iostream>

//Include FLBEM-modules
#include "triangulation.h"
#include "geometrics.h"

using namespace triangulation;
using namespace geometrics;

namespace quadrature{

  //To make the antiderivatives work the rectangles must be given in certain
  //geometric allignement. To express the possible allignements this enum-type
  //is declared here.
  enum parallel_direction {XYXY,YZYZ,ZXZX,XYYZ,XYZX,YZXY,YZZX,ZXXY,ZXYZ,NONE};

  double integrate_rectangle_slp(const Shape&,const Shape&,int,int,int,int);

```

```
38    double integrate_rectangle_dlp(const Shape& r1,const Shape& r2,int j,int k,
39                                   int l,int m);
40
41    //Rectangle must be parallel to xy or to zx
42    double chi(Rectangle& R,Point3D& p);
43    double psi_A(Rectangle& R,Point3D& p);
44    double psi_B(Rectangle& R,Point3D& p);
45    double psi_C(Rectangle& R,Point3D& p);
46    double psi_D(Rectangle& R,Point3D& p);
47
48    //RECTANGLES MUST BE XYXY OR XYZX
49    double gauss_quadrature_rectangle(Rectangle& R1,Rectangle& R2,
50                                      double (*basis_function_1)
51                                      (Rectangle& R,Point3D& p),
52                                      double (*basis_function_2)
53                                      (Rectangle& R,Point3D& p));
54
55
56    //For the convenience of the programmes there functions are declared at this
57    //point. The user though should never need to use them.
58    double eval_F(int k,int l,int m,int n,double p,double x1,double x2,double y1,
59                  double y2,double d1,double d2,double a);
60    double eval_F2(int k,int l,int m,int n,double p,double x1,double x2,double x3,
61                   double y1,double y2,double y3,
62                   double d1,double d2,double d3);
63    double eval_G_kl(int k,int l,double p,double y1,double y2,double x1,double x2,
64                     double a);
65    double eval_g(int k,double p,double y,double x,double a);
66    double eval_G_klm(int k,int l,int m,double p,double y1,double y2,double y3,
67                      double x1,double x2,double x3);
68    double eval_GG(int l,int m,int n,double p,double x2,double y1,double y2,
69                   double x1md1,double d2,double a);
70
71    //This routine needs to be declared in the header, since it is called from other
72    //places than quadrature.cpp too. It returns for given rectangles the
73    //geometrical alignement.
74    parallel_direction check_parallel(const Shape& r1,const Shape& r2);
75  }
76
77  #endif
```

A.4.3 quadrature.cpp

```
1   //************************************************************************//
2   //File:                    quadrature.cpp                                  //
3   //This file implements the routines of quadrature.h naively. This means, that //
4   //to compute teh entry in the stiffness matrix a recursion is done. This      //
5   //causes teh code to be very slow  for elements of hogher polynomial order.   //
6   //A fast and efficient implementation will be released at some point. For the //
7   //moment integral_operators.cpp uses teh lapinteg3 routine from maiprogs      //
8   //************************************************************************//
9   //Author: Samuel Ferraz-Leite                                               //
10  //Date: 07.2007                                                             //
11  //Version 0.6 under development, but functional                             //
12  //************************************************************************//
13
14  //Include standard-libraries
15  #include<cmath>
16  #include<vector>
17  #include<list>
18  #include<iostream>
19
20  //Include FLBEM-modules
21  #include "triangulation.h"
```

```
22 #include "geometrics.h"
23 #include "quadrature.h"
24
25 //Include lapinteg3 from maiprogs
26 extern "C" {
27 #include "maiprogs.h"
28 }
29
30 #define PI acos(-1)
31
32 using namespace triangulation;
33 using namespace geometrics;
34
35 namespace quadrature{
36
37 int sign(double a)
38 {
39   if (a<0) return -1;
40   else if(a>0) return 1;
41   else return 0;
42 }
43
44 //Computes the inner integral for triangles. This routine is not yet
45 //tested, and the user should assume it is buggy.
46 double inner_integrate_triangle_slp(const Triangle& t,const Point3D& x)
47 {
48   double DLP=0.;
49   double SLP=0.;
50   std::vector<Point3D> vertices = t.get_vertices();
51   Vector n=vectorproduct(vertices[1]-vertices[0],vertices[2]-vertices[0]);
52   normalize(n);
53   double K=scalarproduct(n,vertices[0]);
54   double t0=K-scalarproduct(n,x);
55   Point3D M=x+(n*t0);
56   Point3D x_translated=x-M;
57   Point3D A_translated=vertices[0]-M;
58   Point3D B_translated=vertices[1]-M;
59   Point3D C_translated=vertices[2]-M;
60
61   double alpha=0.;
62   Matrix rot_matrix(3);
63   if(norm(Vector(x_translated.x(),x_translated.y(),0.))!=0.)
64   {
65     double cosalpha=scalarproduct(Vector(x_translated.x(),x_translated.y(),0.),
66                                   Vector(0.,1.,0.))
67                                   /norm(Vector(x_translated.x(),
68                                                x_translated.y(),0.));
69     alpha=acos(cosalpha);
70     rot_matrix.set(1,1,cosalpha);
71     rot_matrix.set(1,2,-sin(alpha));
72     rot_matrix.set(1,3,0.);
73     rot_matrix.set(2,1,sin(alpha));
74     rot_matrix.set(2,2,cosalpha);
75     rot_matrix.set(2,3,0.);
76     rot_matrix.set(3,1,0.);
77     rot_matrix.set(3,2,0.);
78     rot_matrix.set(3,3,1.);
79
80     x_translated=MVM(rot_matrix,x_translated);
81     A_translated=MVM(rot_matrix,A_translated);
82     B_translated=MVM(rot_matrix,B_translated);
83     C_translated=MVM(rot_matrix,C_translated);
84   }
```

```
85    if(norm(Vector(0.,x_translated.y(),x_translated.z()))!=0.)
86    {
87      double cosalpha=scalarproduct(Vector(0.,x_translated.y(),x_translated.z()),
88                                    Vector(0.,0.,1.))
89                                    /norm(Vector(0.,x_translated.y(),
90                                                 x_translated.z()));
91      alpha=acos(cosalpha);
92      rot_matrix.set(1,1,1.);
93      rot_matrix.set(1,2,0.);
94      rot_matrix.set(1,3,0.);
95      rot_matrix.set(2,1,0.);
96      rot_matrix.set(2,2,cosalpha);
97      rot_matrix.set(2,3,-sin(alpha));
98      rot_matrix.set(3,1,0.);
99      rot_matrix.set(3,2,sin(alpha));
100     rot_matrix.set(3,3,cosalpha);
101
102     x_translated=MVM(rot_matrix,x_translated);
103     A_translated=MVM(rot_matrix,A_translated);
104     B_translated=MVM(rot_matrix,B_translated);
105     C_translated=MVM(rot_matrix,C_translated);
106   }
107
108   //Correct numerical inequalities!
109   A_translated.set_z(0.);
110   B_translated.set_z(0.);
111   C_translated.set_z(0.);
112   x_translated.set_x(0.);
113   x_translated.set_y(0.);
114
115   double A_gamma;
116   double B_gamma;
117   double C_gamma;
118   double sumDLP;
119
120   Vector sigma_gamma=B_translated-A_translated;
121   normalize(sigma_gamma);
122   Vector nu_gamma=Vector(sigma_gamma.y(),-sigma_gamma.x(),0.);
123   double a_gamma=scalarproduct(A_translated,nu_gamma);
124   double e_gamma=sqrt(x_translated.z()*x_translated.z()+a_gamma*a_gamma);
125   double d_gamma_minus=scalarproduct(A_translated,sigma_gamma);
126   double d_gamma_plus=scalarproduct(B_translated,sigma_gamma);
127   double s_gamma_minus=sqrt(e_gamma*e_gamma+d_gamma_minus*d_gamma_minus);
128   double s_gamma_plus=sqrt(e_gamma*e_gamma+d_gamma_plus*d_gamma_plus);
129   double c_gamma_minus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_minus;
130   double c_gamma_plus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_plus;
131   if(fabs(x_translated.z())>0)
132   {
133     A_gamma=a_gamma*(d_gamma_plus*c_gamma_minus-d_gamma_minus*c_gamma_plus);
134     B_gamma=c_gamma_plus*c_gamma_minus+a_gamma*a_gamma*d_gamma_plus*d_gamma_minus;
135     C_gamma=atan2(B_gamma,A_gamma);
136     sumDLP=C_gamma;
137   }
138   double sumSLP=0.;
139   if(d_gamma_minus>=0 && d_gamma_plus>=0)
140     sumSLP=a_gamma*(-log((s_gamma_plus+d_gamma_plus)/(s_gamma_minus+d_gamma_minus))
          /(4*PI));
141   else if(d_gamma_minus<0 && d_gamma_plus>=0)
142     sumSLP=a_gamma*(-log((s_gamma_minus-d_gamma_minus)*(s_gamma_plus+d_gamma_plus)/
          e_gamma)/(4*PI));
143   else
144     sumSLP=a_gamma*(-log((s_gamma_minus-d_gamma_minus)/(s_gamma_plus-d_gamma_plus))
          /(4*PI));
```

```
145
146   sigma_gamma=C_translated-B_translated;
147   normalize(sigma_gamma);
148   nu_gamma=Vector(sigma_gamma.y(),-sigma_gamma.x(),0.);
149   a_gamma=scalarproduct(B_translated,nu_gamma);
150   e_gamma=sqrt(x_translated.z()*x_translated.z()+a_gamma*a_gamma);
151   d_gamma_minus=scalarproduct(B_translated,sigma_gamma);
152   d_gamma_plus=scalarproduct(C_translated,sigma_gamma);
153   s_gamma_minus=sqrt(e_gamma*e_gamma+d_gamma_minus*d_gamma_minus);
154   s_gamma_plus=sqrt(e_gamma*e_gamma+d_gamma_plus*d_gamma_plus);
155   c_gamma_minus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_minus;
156   c_gamma_plus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_plus;
157   if(fabs(x_translated.z())>0)
158   {
159     A_gamma=a_gamma*(d_gamma_plus*c_gamma_minus-d_gamma_minus*c_gamma_plus);
160     B_gamma=c_gamma_plus*c_gamma_minus+a_gamma*a_gamma*d_gamma_plus*d_gamma_minus;
161     C_gamma=atan2(B_gamma,A_gamma);
162     sumDLP+=C_gamma;
163   }
164
165   if(d_gamma_minus>=0 && d_gamma_plus>=0)
166     sumSLP+=a_gamma*(-log((s_gamma_plus+d_gamma_plus)/(s_gamma_minus+d_gamma_minus))
          /(4*PI));
167   else if(d_gamma_minus<0 && d_gamma_plus>=0)
168     sumSLP+=a_gamma*(-log((s_gamma_minus-d_gamma_minus)*(s_gamma_plus+d_gamma_plus)/
          e_gamma)/(4*PI));
169   else
170     sumSLP+=a_gamma*(-log((s_gamma_minus-d_gamma_minus)/(s_gamma_plus-d_gamma_plus))
          /(4*PI));
171
172   sigma_gamma=A_translated-C_translated;
173   normalize(sigma_gamma);
174   nu_gamma=Vector(sigma_gamma.y(),-sigma_gamma.x(),0.);
175   a_gamma=scalarproduct(C_translated,nu_gamma);
176   e_gamma=sqrt(x_translated.z()*x_translated.z()+a_gamma*a_gamma);
177   d_gamma_minus=scalarproduct(C_translated,sigma_gamma);
178   d_gamma_plus=scalarproduct(A_translated,sigma_gamma);
179   s_gamma_minus=sqrt(e_gamma*e_gamma+d_gamma_minus*d_gamma_minus);
180   s_gamma_plus=sqrt(e_gamma*e_gamma+d_gamma_plus*d_gamma_plus);
181   c_gamma_minus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_minus;
182   c_gamma_plus=e_gamma*e_gamma+fabs(x_translated.z())*s_gamma_plus;
183   if(fabs(x_translated.z())>0)
184   {
185     A_gamma=a_gamma*(d_gamma_plus*c_gamma_minus-d_gamma_minus*c_gamma_plus);
186     B_gamma=c_gamma_plus*c_gamma_minus+a_gamma*a_gamma*d_gamma_plus*d_gamma_minus;
187     C_gamma=atan2(B_gamma,A_gamma);
188     sumDLP+=C_gamma;
189   }
190
191   if(d_gamma_minus>=0 && d_gamma_plus>=0)
192     sumSLP+=a_gamma*(-log((s_gamma_plus+d_gamma_plus)/(s_gamma_minus+d_gamma_minus))
          /(4*PI));
193   else if(d_gamma_minus<0 && d_gamma_plus>=0)
194     sumSLP+=a_gamma*(-log((s_gamma_minus-d_gamma_minus)*(s_gamma_plus+d_gamma_plus)/
          e_gamma)/(4*PI));
195   else
196     sumSLP+=a_gamma*(-log((s_gamma_minus-d_gamma_minus)/(s_gamma_plus-d_gamma_plus))
          /(4*PI));
197
198   if(x_translated.z()>0)
199     DLP=sumDLP/(4.*PI);
200   else if(x_translated.z()<0)
201     DLP=-sumDLP/(4.*PI);
```

```
202
203   if(fabs(x_translated.z())>0)
204     SLP=x_translated.z()*DLP+sumSLP;
205   else
206     SLP=sumSLP;
207   return SLP;
208 }
209
210 //Computes the entrie for triangles semianalytically. This routine is not yet
211 //testes and should be assumed to be buggy.
212 double integrate_triangle_slp(const Triangle& t1,const Triangle& t2,int j,int k,
213                               int l,int m)
214 {
215   if(j!=0 || k!=0 || l!=0 || m!=0)
216   {
217     std::cout<<"Sorry not yet implemented!\n";
218     return 0.;
219   }
220   std::vector<Point3D> points=std::vector<Point3D>(7);
221   std::vector<Point3D> vertices=t1.get_vertices();
222   points[0]=Point3D(vertices[0].x()+(6-sqrt(15))/21*
223                     (vertices[1].x()-vertices[0].x())+
224                     (6-sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
225                     vertices[0].y()+(6-sqrt(15))/21*(vertices[1].y()-
226                                                      vertices[0].y())+
227                     (6-sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
228                     vertices[0].z()+(6-sqrt(15))/21*(vertices[1].z()-
229                                                      vertices[0].z())+
230                     (6-sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
231   points[1]=Point3D(vertices[0].x()+(9+2*sqrt(15))/21*(vertices[1].x()-
232                                                        vertices[0].x())+
233                     (6-sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
234                     vertices[0].y()+(9+2*sqrt(15))/21*(vertices[1].y()-
235                                                        vertices[0].y())+
236                     (6-sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
237                     vertices[0].z()+(9+2*sqrt(15))/21*(vertices[1].z()-
238                                                        vertices[0].z())+
239                     (6-sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
240   points[2]=Point3D(vertices[0].x()+(6-sqrt(15))/21*(vertices[1].x()-
241                                                      vertices[0].x())+
242                     (9+2*sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
243                     vertices[0].y()+(6-sqrt(15))/21*(vertices[1].y()-
244                                                      vertices[0].y())+
245                     (9+2*sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
246                     vertices[0].z()+(6-sqrt(15))/21*(vertices[1].z()-
247                                                      vertices[0].z())+
248                     (9+2*sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
249   points[3]=Point3D(vertices[0].x()+(6+sqrt(15))/21*(vertices[1].x()-
250                                                      vertices[0].x())+
251                     (9-2*sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
252                     vertices[0].y()+(6+sqrt(15))/21*(vertices[1].y()-
253                                                      vertices[0].y())+
254                     (9-2*sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
255                     vertices[0].z()+(6+sqrt(15))/21*(vertices[1].z()-
256                                                      vertices[0].z())+
257                     (9-2*sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
258   points[4]=Point3D(vertices[0].x()+(6+sqrt(15))/21*(vertices[1].x()-
259                                                      vertices[0].x())+
260                     (6+sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
261                     vertices[0].y()+(6+sqrt(15))/21*(vertices[1].y()-
262                                                      vertices[0].y())+
263                     (6+sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
264                     vertices[0].z()+(6+sqrt(15))/21*(vertices[1].z()-
```

```
                                                        vertices[0].z())+
                    (6+sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
  points[5]=Point3D(vertices[0].x()+(9-2*sqrt(15))/21*(vertices[1].x()-
                                                          vertices[0].x())+
                    (6+sqrt(15))/21*(vertices[2].x()-vertices[0].x()),
                    vertices[0].y()+(9-2*sqrt(15))/21*(vertices[1].y()-
                                                          vertices[0].y())+
                    (6+sqrt(15))/21*(vertices[2].y()-vertices[0].y()),
                    vertices[0].z()+(9-2*sqrt(15))/21*(vertices[1].z()-
                                                          vertices[0].z())+
                    (6+sqrt(15))/21*(vertices[2].z()-vertices[0].z()));
  points[6]=Point3D(vertices[0].x()+7./21.*(vertices[1].x()-vertices[0].x())+
      7./21.*(vertices[2].x()-vertices[0].x()),
      vertices[0].y()+7./21.*(vertices[1].y()-vertices[0].y())+
      7./21.*(vertices[2].y()-vertices[0].y()),
      vertices[0].z()+7./21.*(vertices[1].z()-vertices[0].z())+
      7./21.*(vertices[2].z()-vertices[0].z()));
  std::vector<double> weigths=std::vector<double>(7);
  weigths[0]=(155-sqrt(15))/2400.;
  weigths[1]=(155-sqrt(15))/2400.;
  weigths[2]=(155-sqrt(15))/2400.;
  weigths[3]=(155+sqrt(15))/2400.;
  weigths[4]=(155+sqrt(15))/2400.;
  weigths[5]=(155+sqrt(15))/2400.;
  weigths[6]=270./2400.;

  double sum=0.;
  for(int i=0;i<7;++i)
    sum+=weigths[i]*inner_integrate_triangle_slp(t2,points[i]);
  return sqrt(t1.gram_det())*sum;
}
//************************************************************************//
// Implementation of the antiderivatives as specified in the Maishak paper    //
//************************************************************************//
double eval_g(int k,double p,double y,double x,double a)
{
  //check which recurrence formula is to be used.
  if(k+2*p+1!=0 && k>=1)
  {
    //recurrence for k>=2 considering all cases, that one of the
    //factors may be 0
    if(k>=2)
    {
      if(x==0 && k+p==0 && a==0)
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
      if(x==0 && k+p==0 && a!=0)
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
            (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
      if(x==0 && k+p!=0 && a==0)
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
      if(x!=0 && k+p==0 && a==0)
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
            (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
      if((x==0 || k+p==0) && a!=0)//Note, that x!=0 or k+p!=0
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
            (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
      else
        return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)+2*x*(k+p)*
                eval_g(k-1,p,y,x,a)-
                (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
    }
    if(k==1)
    {
```

```
328       if(x==0 || k+p==0)
329         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
330       else
331         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)+2*x*(k+p)*
332                 eval_g(k-1,p,y,x,a))/(k+2.*p+1.);
333     }
334   }
335   if(k+2*p+1==0 && k>=1)
336   {
337     if(k>=2 && x==0)
338       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1)-(k-1)*eval_g(k-2,p+1,y,x,a))/
339           (2.*p+2.);
340     if(k>=2 && x!=0)
341       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1)-(k-1)*eval_g(k-2,p+1,y,x,a))/
342           (2.*p+2.)+
343           x*eval_g(k-1,p,y,x,a);
344     if(k==1 && x==0)
345       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1))/(2.*p+2.);
346     else //k==1 && x!=0
347       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1))/(2.*p+2.)+
348           x*eval_g(k-1,p,y,x,a);
349   }
350   //Now the cases, where k==0!
351   if(p>0 && y-x!=0 && a!=0)
352     return ((y-x)*pow(((y-x)*(y-x)+a*a),p)+2.*p*a*a*eval_g(0,p-1,y,x,a))/
353         (2.*p+1.);
354   if(p>0 && y-x!=0 && a==0)
355     return ((y-x)*pow(((y-x)*(y-x)),p))/(2.*p+1.);
356   if(p>0 && y-x==0 && a!=0)
357     return (2.*p*a*a*eval_g(0,p-1,y,x,a))/(2.*p+1.);
358   if(p>0 && y-x==0 && a==0)
359     return 0.;
360   if(p==0)
361     return y-x;
362   if(p==-1 && a!=0)
363     return atan((y-x)/fabs(a));
364   if(p==-1.5 && a!=0)
365     return (y-x)/(a*a)*1./sqrt((y-x)*(y-x)+a*a);
366   if(p==-0.5 && a!=0)
367       return asinh((y-x)/fabs(a));
368   if(a==0)
369     std::cout<<"Something␣went␣wrong!\n";
370   return 0.;
371 }
372
373 double eval_G_kl(int k,int l,double p,double y1,double y2,double x1,
374                  double x2,double a)
375 {
376   //recurrence to pull down p to -1.5
377   if(p>-1.5 && k+l+2*p+2!=0)
378   {
379     double summand1=0.;
380     double summand2=0.;
381     double summand3=0.;
382     double summand4=0.;
383     double summand5=0.;
384     if(y1!=x1 && (y1!=0 || k==0))
385       summand1=pow(y1,k)*(y1-x1)*eval_g(l,p,y2,x2,
386                                         sqrt((y1-x1)*(y1-x1)+a*a));
387     if(x1!=0 && k!=0)
388       summand2=k*x1*eval_G_kl(k-1,l,p,y1,y2,x1,x2,a);
389     if((y2!=0 || l==0) && y2!=x2)
390       summand3=pow(y2,l)*(y2-x2)*eval_g(k,p,y1,x1,
```

```
391                                             sqrt((y2-x2)*(y2-x2)+a*a));
392     if(x2!=0 && l!=0)
393       summand4=l*x2*eval_G_kl(k,l-1,p,y1,y2,x1,x2,a);
394     if(a!=0 && p!=0)
395       summand5=2*p*a*a*eval_G_kl(k,l,p-1,y1,y2,x1,x2,a);
396     return (summand1+summand2+summand3+summand4+summand5)/(k+l+2*p+2.);
397   }
398   if(p>-1.5 && k+l+2*p+2==0 && a!=0)
399   {
400     double summand1=0.;
401     double summand2=0.;
402     double summand3=0.;
403     if(y1!=0 || k-1==0)
404       summand1=pow(y1,k-1)*eval_g(l,p+1,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
405     if(x1!=0 && k+p!=0)
406       summand2=2*x1*(k+p)*eval_G_kl(k-1,l,p,y1,y2,x1,x2,a);
407     if(k-1!=0)
408     {
409       double sum31=0.;
410       double sum32=0.;
411       double sum33=0.;
412       if(x1!=0 || x2!=0 || a!=0)
413         sum31=(x1*x1+x2*x2+a*a)*eval_G_kl(k-2,l,p,y1,y2,x1,x2,a);
414       if(x2!=0)
415         sum32=-2*x2*eval_G_kl(k-2,l+1,p,y1,y2,x1,x2,a);
416       sum33=eval_G_kl(k-2,l+2,p,y1,y2,x1,x2,a);
417       summand3=-(k-1)*(sum31+sum32+sum33);
418     }
419     return (summand1+summand2+summand3)/(k+2*p+1.);
420   }
421
422   //recurrence to pull down k to 0
423   if(p==-1.5 && k>=1)
424   {
425     double summand1=0.;
426     double summand2=0.;
427     double summand3=0.;
428     if(y1!=0 || k-1==0)
429       summand1=-pow(y1,k-1)*eval_g(l,-0.5,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
430     if(k-1!=0)
431       summand2=(k-1)*eval_G_kl(k-2,l,-0.5,y1,y2,x1,x2,a);
432     if(x1!=0)
433       summand3=x1*eval_G_kl(k-1,l,-1.5,y1,y2,x1,x2,a);
434     return summand1+summand2+summand3;
435   }
436   if(p==-1.5 && l>=2)
437   {
438     double summand1=0.;
439     double summand2=0.;
440     double summand3=0.;
441     if(y1!=x1)
442       summand1=(y1-x1)*eval_g(l-2,-0.5,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
443     if(x2!=0)
444       summand2=2*x2*eval_G_kl(0,l-1,-1.5,y1,y2,x1,x2,a);
445     if(x1!=0 || a!=0)
446       summand3=-(x2*x2+a*a)*eval_G_kl(0,l-2,-1.5,y1,y2,x1,x2,a);
447     return summand1+summand2+summand3;
448   }
449   if(p==-1.5 && l==1)
450   {
451     double summand1=0.;
452     double summand2=0.;
453     summand1=-asinh((y1-x1)/sqrt((y2-x2)*(y2-x2)+a*a));
```

```
454       if(x2!=0)
455         summand2=x2*eval_G_kl(0,0,-1.5,y1,y2,x1,x2,a);
456       return summand1+summand2;
457     }
458     if(a!=0)
459       return sign((y1-x1)*(y2-x2)*a)/(2*a)*
460           acos((-2*(y1-x1)*(y1-x1)*(y2-x2)*(y2-x2))/
461                (((y1-x1)*(y1-x1)+a*a)*((y2-x2)*(y2-x2)+a*a))+1);
462     else return 0.;
463   }
464
465   double eval_G_klm(int k,int l,int m,double p,double y1,double y2,double y3,
466                     double x1,double x2,double x3)
467   {
468     double summand1=0.;
469     double summand2=0.;
470     double summand3=0.;
471     double summand4=0.;
472     double summand5=0.;
473     double summand6=0.;
474     if(y1!=x1 && (y1!=0 || k==0))
475       summand1=pow(y1,k)*(y1-x1)*eval_G_kl(l,m,p,y2,y3,x2,x3,y1-x1);
476     if(x1!=0 && k!=0)
477       summand2=x1*k*eval_G_klm(k-1,l,m,p,y1,y2,y3,x1,x2,x3);
478     if(y2!=x2 && (y2!=0 || l==0))
479       summand3=pow(y2,l)*(y2-x2)*eval_G_kl(k,m,p,y1,y3,x1,x3,y2-x2);
480     if(x2!=0 && l!=0)
481       summand4=x2*l*eval_G_klm(k,l-1,m,p,y1,y2,y3,x1,x2,x3);
482     if(y3!=x3 && (y3!=0 || m==0))
483       summand5=pow(y3,m)*(y3-x3)*eval_G_kl(k,l,p,y1,y2,x1,x2,y3-x3);
484     if(x3!=0 && m!=0)
485       summand6=x3*m*eval_G_klm(k,l,m-1,p,y1,y2,y3,x1,x2,x3);
486     return (summand1+summand2+summand3+summand4+summand5+summand6)/(k+l+m+2*p+3);
487   }
488
489   double eval_GG(int l,int m,int n,double p,double x2,double y1,double y2,
490                  double x1md1,double d2,double a)
491   {
492     double summand1=0.;
493     double summand2=0.;
494     double summand3=0.;
495     if(y2!=0)
496       summand1=pow(y2,n+1)*eval_G_kl(m,l,p,y1,x2,x1md1,y2+d2,a);
497     if(x2!=0 || l==0)
498       summand2=pow(x2,l)*eval_G_kl(m,n+1,p,y1,y2,x1md1,x2-d2,a);
499     if(l>0)
500       summand3=-l*eval_GG(l-1,m,n+1,p,x2,y1,y2,x1md1,d2,a);
501     return (summand1+summand2+summand3)/(n+1.);
502   }
503
504   //************************************************************************//
505   // Implementation of the antiderivatives as specified in the Maishak paper    //
506   //************************************************************************//
507   double eval_g(int k,double p,double y,double x,double a)
508   {
509     //check which recurrence formula is to be used.
510     if(k+2*p+1!=0 && k>=1)
511     {
512       //recurrence for k>=2 considering all cases, that one of the
513       //factors may be 0
514       if(k>=2)
515       {
516         if(x==0 && k+p==0 && a==0)
```

```
517         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
518       if(x==0 && k+p==0 && a!=0)
519         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
520             (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
521       if(x==0 && k+p!=0 && a==0)
522         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
523       if(x!=0 && k+p==0 && a==0)
524         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
525             (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
526       if((x==0 || k+p==0) && a!=0)//Note, that x!=0 or k+p!=0
527         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)-
528             (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
529       else
530         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)+2*x*(k+p)*
531                 eval_g(k-1,p,y,x,a)-
532                 (k-1.)*(x*x+a*a)*eval_g(k-2,p,y,x,a))/(k+2.*p+1.);
533     }
534     if(k==1)
535     {
536       if(x==0 || k+p==0)
537         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1))/(k+2.*p+1.);
538       else
539         return (pow(y,k-1)*pow(((y-x)*(y-x)+a*a),p+1)+2*x*(k+p)*
540                 eval_g(k-1,p,y,x,a))/(k+2.*p+1.);
541     }
542   }
543   if(k+2*p+1==0 && k>=1)
544   {
545     if(k>=2 && x==0)
546       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1)-(k-1)*eval_g(k-2,p+1,y,x,a))/
547           (2.*p+2.);
548     if(k>=2 && x!=0)
549       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1)-(k-1)*eval_g(k-2,p+1,y,x,a))/
550           (2.*p+2.)+
551           x*eval_g(k-1,p,y,x,a);
552     if(k==1 && x==0)
553       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1))/(2.*p+2.);
554     else //k==1 && x!=0
555       return (pow(y,k-1)*pow((y-x)*(y-x)+a*a,p+1))/(2.*p+2.)+
556           x*eval_g(k-1,p,y,x,a);
557   }
558   //Now the cases, where k==0!
559   if(p>0 && y-x!=0 && a!=0)
560     return ((y-x)*pow(((y-x)*(y-x)+a*a),p)+2.*p*a*a*eval_g(0,p-1,y,x,a))/
561         (2.*p+1.);
562   if(p>0 && y-x!=0 && a==0)
563     return ((y-x)*pow(((y-x)*(y-x)),p))/(2.*p+1.);
564   if(p>0 && y-x==0 && a!=0)
565     return (2.*p*a*a*eval_g(0,p-1,y,x,a))/(2.*p+1.);
566   if(p>0 && y-x==0 && a==0)
567     return 0.;
568   if(p==0)
569     return y-x;
570   if(p==-1 && a!=0)
571     return atan((y-x)/fabs(a));
572   if(p==-1.5 && a!=0)
573     return (y-x)/(a*a)*1./sqrt((y-x)*(y-x)+a*a);
574   if(p==-0.5 && a!=0)
575       return asinh((y-x)/fabs(a));
576   if(a==0)
577     std::cout<<"Something␣went␣wrong!\n";
578   return 0.;
579 }
```

```
580
581 double eval_G_kl(int k,int l,double p,double y1,double y2,double x1,
582                   double x2,double a)
583 {
584   //recurrence to pull down p to -1.5
585   if(p>-1.5 && k+l+2*p+2!=0)
586   {
587     double summand1=0.;
588     double summand2=0.;
589     double summand3=0.;
590     double summand4=0.;
591     double summand5=0.;
592     if(y1!=x1 && (y1!=0 || k==0))
593       summand1=pow(y1,k)*(y1-x1)*eval_g(l,p,y2,x2,
594                                         sqrt((y1-x1)*(y1-x1)+a*a));
595     if(x1!=0 && k!=0)
596       summand2=k*x1*eval_G_kl(k-1,l,p,y1,y2,x1,x2,a);
597     if((y2!=0 || l==0) && y2!=x2)
598       summand3=pow(y2,l)*(y2-x2)*eval_g(k,p,y1,x1,
599                                         sqrt((y2-x2)*(y2-x2)+a*a));
600     if(x2!=0 && l!=0)
601       summand4=l*x2*eval_G_kl(k,l-1,p,y1,y2,x1,x2,a);
602     if(a!=0 && p!=0)
603       summand5=2*p*a*a*eval_G_kl(k,l,p-1,y1,y2,x1,x2,a);
604     return (summand1+summand2+summand3+summand4+summand5)/(k+l+2*p+2.);
605   }
606   if(p>-1.5 && k+l+2*p+2==0 && a!=0)
607   {
608     double summand1=0.;
609     double summand2=0.;
610     double summand3=0.;
611     if(y1!=0 || k-1==0)
612       summand1=pow(y1,k-1)*eval_g(l,p+1,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
613     if(x1!=0 && k+p!=0)
614       summand2=2*x1*(k+p)*eval_G_kl(k-1,l,p,y1,y2,x1,x2,a);
615     if(k-1!=0)
616     {
617       double sum31=0.;
618       double sum32=0.;
619       double sum33=0.;
620       if(x1!=0 || x2!=0 || a!=0)
621         sum31=(x1*x1+x2*x2+a*a)*eval_G_kl(k-2,l,p,y1,y2,x1,x2,a);
622       if(x2!=0)
623         sum32=-2*x2*eval_G_kl(k-2,l+1,p,y1,y2,x1,x2,a);
624       sum33=eval_G_kl(k-2,l+2,p,y1,y2,x1,x2,a);
625       summand3=-(k-1)*(sum31+sum32+sum33);
626     }
627     return (summand1+summand2+summand3)/(k+2*p+1.);
628   }
629
630   //recurrence to pull down k to 0
631   if(p==-1.5 && k>=1)
632   {
633     double summand1=0.;
634     double summand2=0.;
635     double summand3=0.;
636     if(y1!=0 || k-1==0)
637       summand1=-pow(y1,k-1)*eval_g(l,-0.5,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
638     if(k-1!=0)
639       summand2=(k-1)*eval_G_kl(k-2,l,-0.5,y1,y2,x1,x2,a);
640     if(x1!=0)
641       summand3=x1*eval_G_kl(k-1,l,-1.5,y1,y2,x1,x2,a);
642     return summand1+summand2+summand3;
```

```
643   }
644   if(p==-1.5 && l>=2)
645   {
646     double summand1=0.;
647     double summand2=0.;
648     double summand3=0.;
649     if(y1!=x1)
650       summand1=(y1-x1)*eval_g(l-2,-0.5,y2,x2,sqrt((y1-x1)*(y1-x1)+a*a));
651     if(x2!=0)
652       summand2=2*x2*eval_G_kl(0,l-1,-1.5,y1,y2,x1,x2,a);
653     if(x1!=0 || a!=0)
654       summand3=-(x2*x2+a*a)*eval_G_kl(0,l-2,-1.5,y1,y2,x1,x2,a);
655     return summand1+summand2+summand3;
656   }
657   if(p==-1.5 && l==1)
658   {
659     double summand1=0.;
660     double summand2=0.;
661     summand1=-asinh((y1-x1)/sqrt((y2-x2)*(y2-x2)+a*a));
662     if(x2!=0)
663       summand2=x2*eval_G_kl(0,0,-1.5,y1,y2,x1,x2,a);
664     return summand1+summand2;
665   }
666   if(a!=0)
667     return sign((y1-x1)*(y2-x2)*a)/(2*a)*
668         acos((-2*(y1-x1)*(y1-x1)*(y2-x2)*(y2-x2))/
669              (((y1-x1)*(y1-x1)+a*a)*((y2-x2)*(y2-x2)+a*a))+1);
670   else return 0.;
671 }
672
673 double eval_G_klm(int k,int l,int m,double p,double y1,double y2,double y3,
674                   double x1,double x2,double x3)
675 {
676   double summand1=0.;
677   double summand2=0.;
678   double summand3=0.;
679   double summand4=0.;
680   double summand5=0.;
681   double summand6=0.;
682   if(y1!=x1 && (y1!=0 || k==0))
683     summand1=pow(y1,k)*(y1-x1)*eval_G_kl(l,m,p,y2,y3,x2,x3,y1-x1);
684   if(x1!=0 && k!=0)
685     summand2=x1*k*eval_G_klm(k-1,l,m,p,y1,y2,y3,x1,x2,x3);
686   if(y2!=x2 && (y2!=0 || l==0))
687     summand3=pow(y2,l)*(y2-x2)*eval_G_kl(k,m,p,y1,y3,x1,x3,y2-x2);
688   if(x2!=0 && l!=0)
689     summand4=x2*l*eval_G_klm(k,l-1,m,p,y1,y2,y3,x1,x2,x3);
690   if(y3!=x3 && (y3!=0 || m==0))
691     summand5=pow(y3,m)*(y3-x3)*eval_G_kl(k,l,p,y1,y2,x1,x2,y3-x3);
692   if(x3!=0 && m!=0)
693     summand6=x3*m*eval_G_klm(k,l,m-1,p,y1,y2,y3,x1,x2,x3);
694   return (summand1+summand2+summand3+summand4+summand5+summand6)/(k+l+m+2*p+3);
695 }
696
697 double eval_GG(int l,int m,int n,double p,double x2,double y1,double y2,
698                double x1md1,double d2,double a)
699 {
700   double summand1=0.;
701   double summand2=0.;
702   double summand3=0.;
703   if(y2!=0)
704     summand1=pow(y2,n+1)*eval_G_kl(m,l,p,y1,x2,x1md1,y2+d2,a);
705   if(x2!=0 || l==0)
```

```
706     summand2=pow(x2,l)*eval_G_kl(m,n+1,p,y1,y2,x1md1,x2-d2,a);
707   if(l>0)
708     summand3=-l*eval_GG(l-1,m,n+1,p,x2,y1,y2,x1md1,d2,a);
709   return (summand1+summand2+summand3)/(n+1.);
710 }
711
712 double eval_F(int k,int l,int m,int n,double p,double x1,double x2,double y1,
713               double y2,double d1,double d2,double a)
714 {
715   if(k==0 && l==0 && m==0 && n==0)
716   {
717     if(p>-1.5)
718     {
719       double summand1=0.;
720       double summand2=0.;
721       double summand3=0.;
722       double summand4=0.;
723       if(x2-y2-d2!=0)
724         summand1=(x2-y2-d2)*eval_g(0,p+1,x2,y2+d2,
725                                    sqrt((x1-y1-d1)*(x1-y1-d1)+a*a));
726       if(x1-y1-d1!=0)
727         summand2=(x1-y1-d1)*eval_g(0,p+1,x1,y1+d1,
728                                    sqrt((x2-y2-d2)*(x2-y2-d2)+a*a));
729       summand3=-1./(p+2.)*pow((x1-y1-d1)*(x1-y1-d1)+
730                               (x2-y2-d2)*(x2-y2-d2)+a*a,p+2);
731       if(a!=0)
732         summand4=a*a*2*p/(2*p+2)*eval_F(0,0,0,0,p-1,x1,x2,y1,y2,d1,d2,a);
733       return 1./(2.*p+2.)*(summand1+summand2+summand3)+summand4;
734     }
735     if(p==-1.5)
736     {
737       double summand1=0.;
738       double summand2=0.;
739       double summand3=0.;
740       double summand4=0.;
741       if(x1-y1-d1!=0 && x2-y2-d2!=0)
742         summand1=(x1-y1-d1)*(x2-y2-d2)*eval_G_kl(0,0,p,x1,x2,y1+d1,y2+d2,a);
743       if(x1-y1-d1!=0)
744         summand2=-1./(2.*p+2.)*(x1-y1-d1)*eval_g(0,p+1,x1,y1+d1,
745                                                  sqrt((x2-y2-d2)*(x2-y2-d2)+a*a));
746       if(x2-y2-d2!=0)
747         summand3=-1./(2.*p+2.)*(x2-y2-d2)*eval_g(0,p+1,x2,y2+d2,
748                                                  sqrt((x1-y1-d1)*(x1-y1-d1)+a*a));
749       summand4=1./((2.*p+2.)*(2.*p+4.))*pow((x1-y1-d1)*(x1-y1-d1)+
750                                             (x2-y2-d2)*(x2-y2-d2)+a*a,p+2);
751       return summand1+summand2+summand3+summand4;
752     }
753   }
754   else
755   {
756     double summand1=0.;
757     double summand2=0.;
758     double summand3=0.;
759     if(y1!=0)
760       summand1=pow(y1,m+1)*eval_GG(l,k,n,p,x2,x1,y2,y1+d1,d2,a);
761     if(x1!=0 || k==0)
762       summand2=pow(x1,k)*eval_GG(l,m+1,n,p,x2,y1,y2,x1-d1,d2,a);
763     if(k>0)
764       summand3=-k*eval_F(k-1,l,m+1,n,p,x1,x2,y1,y2,d1,d2,a);
765     return 1./(double)(m+1.)*(summand1+summand2+summand3);
766   }
767   return 0.;
768 }
```

```
769
770 double eval_F2(int k,int l,int m,int n,double p,double x1,double x2,double x3,
771                 double y1,double y2,double y3,double d1,double d2,double d3)
772 {
773   if(k==0 && l==0 && m==0 && n==0)
774   {
775     double summand1=0.;
776     double summand2=0.;
777     double summand3=0.;
778     double summand4=0.;
779     double summand5=0.;
780     summand1=-1./(2.*p+2.)*eval_G_kl(0,0,p+1,y3,x1,x3-d3,y1+d1,x2-y2);
781     if(x1-y1-d1!=0 && x2-y2-d2!=0)
782       summand2=-(x1-y1-d1)*(x2-y2-d2)*eval_G_kl(0,0,p,x2,y3,y2+d2,x3-d3,x1-y1-d1);
783     if(x1-y1-d1!=0)
784       summand3=(x1-y1-d1)/(2.*p+2.)*eval_g(0,p+1,y3,x3-d3,
785                                            sqrt((x1-y1-d1)*(x1-y1-d1)+
786                                                 (x2-y2-d2)*(x2-y2-d2)));
787     if(x3-y3-d3!=0 && x2-y2-d2!=0)
788       summand4=(x3-y3-d3)*(x2-y2-d2)*eval_G_kl(0,0,p,x1,x2,y1+d1,y2+d2,x3-y3-d3);
789     if(x3-y3-d3!=0)
790       summand5=-(x3-y3-d3)/(2.*p+2.)*eval_g(0,p+1,x1,y1+d1,
791                                             sqrt((x2-y2-d2)*(x2-y2-d2)+
792                                                  (x3-y3-d3)*(x3-y3-d3)));
793     return (summand1+summand2+summand3+summand4+summand5)/(2.*p+3.);
794   }
795   else
796   {
797     double summand1=0.;
798     double summand2=0.;
799     double summand3=0.;
800     if(y2!=0)
801       summand1=pow(y2,m+1)*eval_G_klm(k,l,n,p,x1,x2,y3,y1+d1,y2+d2,x3-d3);
802     if(x2!=0 || l==0)
803       summand2=pow(x2,l)*eval_G_klm(k,m+1,n,p,x1,y2,y3,y1+d1,x2-d2,x3-d3);
804     if(l!=0)
805       summand3=l*eval_F2(k,l-1,m+1,n,p,x1,x2,x3,y1,y2,y3,d1,d2,d3);
806     return (summand1+summand2-summand3)/(m+1.);
807   }
808 }
809
810 double eval_K(int k,int l,int m,int n,double n3,double p,double x1,double x2,
811               double x3,double y1,double y2,double y3,double d1,double d2,
812               double d3)
813 {
814   double summand=0.;
815   double a=x3-y3-d3;
816   if(n3!=0 && a!=0)
817     summand=-n3*a*2*p*eval_F(k,l,m,n,p-1,x1,x2,y1,y2,d1,d2,a);
818   return summand;
819 }
820
821 double eval_K2(int k,int l,int m,int n,double n3,double p,double x1,double x2,
822                double x3,double y1,double y2,double y3,double d1,double d2,
823                double d3)
824 {
825   double summand1=0.;
826   double summand2=0.;
827   if(n3!=0 && (x3!=0 || l==0))
828     summand1=-n3*pow(x3,l)*eval_GG(k,m,n,p,x2,y1,y2,x1-d1,d2,x3-y3-d3);
829   if(l!=0 && n3!=0)
830     summand2=l*n3*eval_F2(l-1,k,n,m,p,x3,x2,x1,y3,y2,y1,d3,d2,d1);
831   return summand1+summand2;
```

```
832 }
833
834 //*************************************************************************//
835 // Implementation of other functions, such as integrate_rectangle_slp      //
836 //*************************************************************************//
837 parallel_direction check_parallel(const Shape& r1,const Shape& r2)
838 {
839   std::vector<Point3D> v1=r1.get_vertices();
840   std::vector<Point3D> v2=r2.get_vertices();
841
842   if(v1[0].z()==v1[2].z())
843   {
844     if(v2[0].z()==v2[2].z())
845       return XYXY;
846     if(v2[0].y()==v2[2].y())
847       return XYZX;
848     if(v2[0].x()==v2[2].x())
849       return XYYZ;
850   }
851   if(v1[0].x()==v1[2].x())
852   {
853     if(v2[0].z()==v2[2].z())
854       return YZXY;
855     if(v2[0].y()==v2[2].y())
856       return YZZX;
857     if(v2[0].x()==v2[2].x())
858       return YZYZ;
859   }
860   if(v1[0].y()==v1[2].y())
861   {
862     if(v2[0].z()==v2[2].z())
863       return ZXXY;
864     if(v2[0].y()==v2[2].y())
865       return ZXZX;
866     if(v2[0].x()==v2[2].x())
867       return ZXYZ;
868   }
869
870   return NONE;
871 }
872
873 double integrate_rectangle_slp(const Shape& r1,const Shape& r2,int j,
874                                int k,int l,int m)
875 {
876   double result=0.;
877   double x1upper=0.;
878   double x1lower=0.;
879   double x2upper=0.;
880   double x2lower=0.;
881   double y1upper=0.;
882   double y1lower=0.;
883   double y2upper=0.;
884   double y2lower=0.;
885   double a=0.;
886   double x3=0.;
887   double y3=0.;
888   bool parallel=false;
889   Matrix rot=Matrix(3);
890   std::vector<Point3D> v1=r1.get_vertices();
891   std::vector<Point3D> v2=r2.get_vertices();
892   switch (check_parallel(r1,r2))
893   {
894     case XYXY:
```

```
895         parallel=true;
896         x1upper=v1[0].x();
897         x1lower=v1[0].x();
898         x2upper=v1[0].y();
899         x2lower=v1[0].y();
900         y1upper=v2[0].x();
901         y1lower=v2[0].x();
902         y2upper=v2[0].y();
903         y2lower=v2[0].y();
904         a=v1[0].z()-v2[0].z();
905         for(int i=1;i<4;++i)
906         {
907           if(v1[i].x()>x1upper)
908             x1upper=v1[i].x();
909           if(v1[i].x()<x1lower)
910             x1lower=v1[i].x();
911           if(v1[i].y()>x2upper)
912             x2upper=v1[i].y();
913           if(v1[i].y()<x2lower)
914             x2lower=v1[i].y();
915           if(v2[i].x()>y1upper)
916             y1upper=v2[i].x();
917           if(v2[i].x()<y1lower)
918             y1lower=v2[i].x();
919           if(v2[i].y()>y2upper)
920             y2upper=v2[i].y();
921           if(v2[i].y()<y2lower)
922             y2lower=v2[i].y();
923         }
924         break;
925       case XYYZ:
926         x1upper=v1[0].x();
927         x1lower=v1[0].x();
928         x2upper=v1[0].y();
929         x2lower=v1[0].y();
930         y1upper=v2[0].y();
931         y1lower=v2[0].y();
932         y2upper=v2[0].z();
933         y2lower=v2[0].z();
934         x3=v1[0].z();
935         y3=v2[0].x();
936         for(int i=1;i<4;++i)
937         {
938           if(v1[i].x()>x1upper)
939             x1upper=v1[i].x();
940           if(v1[i].x()<x1lower)
941             x1lower=v1[i].x();
942           if(v1[i].y()>x2upper)
943             x2upper=v1[i].y();
944           if(v1[i].y()<x2lower)
945             x2lower=v1[i].y();
946           if(v2[i].y()>y1upper)
947             y1upper=v2[i].y();
948           if(v2[i].y()<y1lower)
949             y1lower=v2[i].y();
950           if(v2[i].z()>y2upper)
951             y2upper=v2[i].z();
952           if(v2[i].z()<y2lower)
953             y2lower=v2[i].z();
954         }
955         break;
956       case XYZX:
957         rot.set(1,1,0);
```

```
958         rot.set(1,2,-1);
959         rot.set(1,3,0);
960         rot.set(2,1,1);
961         rot.set(2,2,0);
962         rot.set(2,3,0);
963         rot.set(3,1,0);
964         rot.set(3,2,0);
965         rot.set(3,3,1);
966         for(int i=0;i<4;++i)
967         {
968           v1[i]=MVM(rot,v1[i]);
969           v2[i]=MVM(rot,v2[i]);
970         }
971         x1upper=v1[0].x();
972         x1lower=v1[0].x();
973         x2upper=v1[0].y();
974         x2lower=v1[0].y();
975         y1upper=v2[0].y();
976         y1lower=v2[0].y();
977         y2upper=v2[0].z();
978         y2lower=v2[0].z();
979         x3=v1[0].z();
980         y3=v2[0].x();
981         for(int i=1;i<4;++i)
982         {
983           if(v1[i].x()>x1upper)
984             x1upper=v1[i].x();
985           if(v1[i].x()<x1lower)
986             x1lower=v1[i].x();
987           if(v1[i].y()>x2upper)
988             x2upper=v1[i].y();
989           if(v1[i].y()<x2lower)
990             x2lower=v1[i].y();
991           if(v2[i].y()>y1upper)
992             y1upper=v2[i].y();
993           if(v2[i].y()<y1lower)
994             y1lower=v2[i].y();
995           if(v2[i].z()>y2upper)
996             y2upper=v2[i].z();
997           if(v2[i].z()<y2lower)
998             y2lower=v2[i].z();
999         }
1000        break;
1001      case YZYZ:
1002        parallel=true;
1003        x1upper=v1[0].y();
1004        x1lower=v1[0].y();
1005        x2upper=v1[0].z();
1006        x2lower=v1[0].z();
1007        y1upper=v2[0].y();
1008        y1lower=v2[0].y();
1009        y2upper=v2[0].z();
1010        y2lower=v2[0].z();
1011        a=v1[0].x()-v2[0].x();
1012        for(int i=1;i<4;++i)
1013        {
1014          if(v1[i].y()>x1upper)
1015            x1upper=v1[i].y();
1016          if(v1[i].y()<x1lower)
1017            x1lower=v1[i].y();
1018          if(v1[i].z()>x2upper)
1019            x2upper=v1[i].z();
1020          if(v1[i].z()<x2lower)
```

```
1021            x2lower=v1[i].z();
1022          if(v2[i].y()>y1upper)
1023            y1upper=v2[i].y();
1024          if(v2[i].y()<y1lower)
1025            y1lower=v2[i].y();
1026          if(v2[i].z()>y2upper)
1027            y2upper=v2[i].z();
1028          if(v2[i].z()<y2lower)
1029            y2lower=v2[i].z();
1030        }
1031        break;
1032      case YZXY:
1033        x1upper=v2[0].x();
1034        x1lower=v2[0].x();
1035        x2upper=v2[0].y();
1036        x2lower=v2[0].y();
1037        y1upper=v1[0].y();
1038        y1lower=v1[0].y();
1039        y2upper=v1[0].z();
1040        y2lower=v1[0].z();
1041        x3=v2[0].z();
1042        y3=v1[0].x();
1043        for(int i=1;i<4;++i)
1044        {
1045          if(v2[i].x()>x1upper)
1046            x1upper=v2[i].x();
1047          if(v2[i].x()<x1lower)
1048            x1lower=v2[i].x();
1049          if(v2[i].y()>x2upper)
1050            x2upper=v2[i].y();
1051          if(v2[i].y()<x2lower)
1052            x2lower=v2[i].y();
1053          if(v1[i].y()>y1upper)
1054            y1upper=v1[i].y();
1055          if(v1[i].y()<y1lower)
1056            y1lower=v1[i].y();
1057          if(v1[i].z()>y2upper)
1058            y2upper=v1[i].z();
1059          if(v1[i].z()<y2lower)
1060            y2lower=v1[i].z();
1061        }
1062        break;
1063      case YZZX:
1064        rot.set(1,1,0);
1065        rot.set(1,2,0);
1066        rot.set(1,3,1);
1067        rot.set(2,1,0);
1068        rot.set(2,2,1);
1069        rot.set(2,3,0);
1070        rot.set(3,1,-1);
1071        rot.set(3,2,0);
1072        rot.set(3,3,0);
1073        for(int i=0;i<4;++i)
1074        {
1075          v1[i]=MVM(rot,v1[i]);
1076          v2[i]=MVM(rot,v2[i]);
1077        }
1078        rot.set(1,1,0);
1079        rot.set(1,2,-1);
1080        rot.set(1,3,0);
1081        rot.set(2,1,1);
1082        rot.set(2,2,0);
1083        rot.set(2,3,0);
```

```
1084        rot.set(3,1,0);
1085        rot.set(3,2,0);
1086        rot.set(3,3,1);
1087        for(int i=0;i<4;++i)
1088        {
1089          v1[i]=MVM(rot,v1[i]);
1090          v2[i]=MVM(rot,v2[i]);
1091        }
1092        x1upper=v1[0].x();
1093        x1lower=v1[0].x();
1094        x2upper=v1[0].y();
1095        x2lower=v1[0].y();
1096        y1upper=v2[0].y();
1097        y1lower=v2[0].y();
1098        y2upper=v2[0].z();
1099        y2lower=v2[0].z();
1100        x3=v1[0].z();
1101        y3=v2[0].x();
1102        for(int i=1;i<4;++i)
1103        {
1104          if(v1[i].x()>x1upper)
1105            x1upper=v1[i].x();
1106          if(v1[i].x()<x1lower)
1107            x1lower=v1[i].x();
1108          if(v1[i].y()>x2upper)
1109            x2upper=v1[i].y();
1110          if(v1[i].y()<x2lower)
1111            x2lower=v1[i].y();
1112          if(v2[i].y()>y1upper)
1113            y1upper=v2[i].y();
1114          if(v2[i].y()<y1lower)
1115            y1lower=v2[i].y();
1116          if(v2[i].z()>y2upper)
1117            y2upper=v2[i].z();
1118          if(v2[i].z()<y2lower)
1119            y2lower=v2[i].z();
1120        }
1121        break;
1122      case ZXZX:
1123        parallel=true;
1124        x1upper=v1[0].x();
1125        x1lower=v1[0].x();
1126        x2upper=v1[0].z();
1127        x2lower=v1[0].z();
1128        y1upper=v2[0].x();
1129        y1lower=v2[0].x();
1130        y2upper=v2[0].z();
1131        y2lower=v2[0].z();
1132        a=v1[0].y()-v2[0].y();
1133        for(int i=1;i<4;++i)
1134        {
1135          if(v1[i].x()>x1upper)
1136            x1upper=v1[i].x();
1137          if(v1[i].x()<x1lower)
1138            x1lower=v1[i].x();
1139          if(v1[i].z()>x2upper)
1140            x2upper=v1[i].z();
1141          if(v1[i].z()<x2lower)
1142            x2lower=v1[i].z();
1143          if(v2[i].x()>y1upper)
1144            y1upper=v2[i].x();
1145          if(v2[i].x()<y1lower)
1146            y1lower=v2[i].x();
```

```
1147          if(v2[i].z()>y2upper)
1148            y2upper=v2[i].z();
1149          if(v2[i].z()<y2lower)
1150            y2lower=v2[i].z();
1151        }
1152        break;
1153      case ZXXY:
1154        rot.set(1,1,0);
1155        rot.set(1,2,-1);
1156        rot.set(1,3,0);
1157        rot.set(2,1,1);
1158        rot.set(2,2,0);
1159        rot.set(2,3,0);
1160        rot.set(3,1,0);
1161        rot.set(3,2,0);
1162        rot.set(3,3,1);
1163        for(int i=0;i<4;++i)
1164        {
1165          v1[i]=MVM(rot,v1[i]);
1166          v2[i]=MVM(rot,v2[i]);
1167        }
1168        x1upper=v2[0].x();
1169        x1lower=v2[0].x();
1170        x2upper=v2[0].y();
1171        x2lower=v2[0].y();
1172        y1upper=v1[0].y();
1173        y1lower=v1[0].y();
1174        y2upper=v1[0].z();
1175        y2lower=v1[0].z();
1176        x3=v2[0].z();
1177        y3=v1[0].x();
1178        for(int i=1;i<4;++i)
1179        {
1180          if(v2[i].x()>x1upper)
1181            x1upper=v2[i].x();
1182          if(v2[i].x()<x1lower)
1183            x1lower=v2[i].x();
1184          if(v2[i].y()>x2upper)
1185            x2upper=v2[i].y();
1186          if(v2[i].y()<x2lower)
1187            x2lower=v2[i].y();
1188          if(v1[i].y()>y1upper)
1189            y1upper=v1[i].y();
1190          if(v1[i].y()<y1lower)
1191            y1lower=v1[i].y();
1192          if(v1[i].z()>y2upper)
1193            y2upper=v1[i].z();
1194          if(v1[i].z()<y2lower)
1195            y2lower=v1[i].z();
1196        }
1197        break;
1198      case ZXYZ:
1199        rot.set(1,1,0);
1200        rot.set(1,2,0);
1201        rot.set(1,3,1);
1202        rot.set(2,1,0);
1203        rot.set(2,2,1);
1204        rot.set(2,3,0);
1205        rot.set(3,1,-1);
1206        rot.set(3,2,0);
1207        rot.set(3,3,0);
1208        for(int i=0;i<4;++i)
1209        {
```

```
1210          v1[i]=MVM(rot,v1[i]);
1211          v2[i]=MVM(rot,v2[i]);
1212        }
1213        rot.set(1,1,0);
1214        rot.set(1,2,-1);
1215        rot.set(1,3,0);
1216        rot.set(2,1,1);
1217        rot.set(2,2,0);
1218        rot.set(2,3,0);
1219        rot.set(3,1,0);
1220        rot.set(3,2,0);
1221        rot.set(3,3,1);
1222        for(int i=0;i<4;++i)
1223        {
1224          v1[i]=MVM(rot,v1[i]);
1225          v2[i]=MVM(rot,v2[i]);
1226        }
1227        x1upper=v2[0].x();
1228        x1lower=v2[0].x();
1229        x2upper=v2[0].y();
1230        x2lower=v2[0].y();
1231        y1upper=v1[0].y();
1232        y1lower=v1[0].y();
1233        y2upper=v1[0].z();
1234        y2lower=v1[0].z();
1235        x3=v2[0].z();
1236        y3=v1[0].x();
1237        for(int i=1;i<4;++i)
1238        {
1239          if(v2[i].x()>x1upper)
1240            x1upper=v2[i].x();
1241          if(v2[i].x()<x1lower)
1242            x1lower=v2[i].x();
1243          if(v2[i].y()>x2upper)
1244            x2upper=v2[i].y();
1245          if(v2[i].y()<x2lower)
1246            x2lower=v2[i].y();
1247          if(v1[i].y()>y1upper)
1248            y1upper=v1[i].y();
1249          if(v1[i].y()<y1lower)
1250            y1lower=v1[i].y();
1251          if(v1[i].z()>y2upper)
1252            y2upper=v1[i].z();
1253          if(v1[i].z()<y2lower)
1254            y2lower=v1[i].z();
1255        }
1256        break;
1257      default:
1258        break;
1259    }
1260    if (parallel)
1261    {
1262      result=eval_F(j,k,l,m,-0.5,x1upper,x2upper,y1upper,y2upper,0,0,a)
1263          -eval_F(j,k,l,m,-0.5,x1upper,x2upper,y1upper,y2lower,0,0,a)
1264          -eval_F(j,k,l,m,-0.5,x1upper,x2upper,y1lower,y2upper,0,0,a)
1265          +eval_F(j,k,l,m,-0.5,x1upper,x2upper,y1lower,y2lower,0,0,a)
1266          -eval_F(j,k,l,m,-0.5,x1upper,x2lower,y1upper,y2upper,0,0,a)
1267          +eval_F(j,k,l,m,-0.5,x1upper,x2lower,y1upper,y2lower,0,0,a)
1268          +eval_F(j,k,l,m,-0.5,x1upper,x2lower,y1lower,y2upper,0,0,a)
1269          -eval_F(j,k,l,m,-0.5,x1upper,x2lower,y1lower,y2lower,0,0,a)
1270          -eval_F(j,k,l,m,-0.5,x1lower,x2upper,y1upper,y2upper,0,0,a)
1271          +eval_F(j,k,l,m,-0.5,x1lower,x2upper,y1upper,y2lower,0,0,a)
1272          +eval_F(j,k,l,m,-0.5,x1lower,x2upper,y1lower,y2upper,0,0,a)
```

```
1273          -eval_F(j,k,l,m,-0.5,x1lower,x2upper,y1lower,y2lower,0,0,a)
1274          +eval_F(j,k,l,m,-0.5,x1lower,x2lower,y1upper,y2upper,0,0,a)
1275          -eval_F(j,k,l,m,-0.5,x1lower,x2lower,y1upper,y2lower,0,0,a)
1276          -eval_F(j,k,l,m,-0.5,x1lower,x2lower,y1lower,y2upper,0,0,a)
1277          +eval_F(j,k,l,m,-0.5,x1lower,x2lower,y1lower,y2lower,0,0,a);
1278    }
1279    else
1280    {
1281      result=eval_F2(j,k,l,m,-0.5,x1upper,x2upper,x3,y3,y1upper,y2upper,0,0,0)
1282          -eval_F2(j,k,l,m,-0.5,x1upper,x2upper,x3,y3,y1upper,y2lower,0,0,0)
1283          -eval_F2(j,k,l,m,-0.5,x1upper,x2upper,x3,y3,y1lower,y2upper,0,0,0)
1284          +eval_F2(j,k,l,m,-0.5,x1upper,x2upper,x3,y3,y1lower,y2lower,0,0,0)
1285          -eval_F2(j,k,l,m,-0.5,x1upper,x2lower,x3,y3,y1upper,y2upper,0,0,0)
1286          +eval_F2(j,k,l,m,-0.5,x1upper,x2lower,x3,y3,y1upper,y2lower,0,0,0)
1287          +eval_F2(j,k,l,m,-0.5,x1upper,x2lower,x3,y3,y1lower,y2upper,0,0,0)
1288          -eval_F2(j,k,l,m,-0.5,x1upper,x2lower,x3,y3,y1lower,y2lower,0,0,0)
1289          -eval_F2(j,k,l,m,-0.5,x1lower,x2upper,x3,y3,y1upper,y2upper,0,0,0)
1290          +eval_F2(j,k,l,m,-0.5,x1lower,x2upper,x3,y3,y1upper,y2lower,0,0,0)
1291          +eval_F2(j,k,l,m,-0.5,x1lower,x2upper,x3,y3,y1lower,y2upper,0,0,0)
1292          -eval_F2(j,k,l,m,-0.5,x1lower,x2upper,x3,y3,y1lower,y2lower,0,0,0)
1293          +eval_F2(j,k,l,m,-0.5,x1lower,x2lower,x3,y3,y1upper,y2upper,0,0,0)
1294          -eval_F2(j,k,l,m,-0.5,x1lower,x2lower,x3,y3,y1upper,y2lower,0,0,0)
1295          -eval_F2(j,k,l,m,-0.5,x1lower,x2lower,x3,y3,y1lower,y2upper,0,0,0)
1296          +eval_F2(j,k,l,m,-0.5,x1lower,x2lower,x3,y3,y1lower,y2lower,0,0,0);
1297    }
1298    return result/(4*PI);
1299  }
1300
1301  //Computes the entry into teh stiffness matrix for the DLP
1302  double integrate_rectangle_dlp(const Shape& r1,const Shape& r2,int j,int k,
1303                                 int l,int m)
1304  {
1305    double result=0.;
1306    double x1upper=0.;
1307    double x1lower=0.;
1308    double x2upper=0.;
1309    double x2lower=0.;
1310    double y1upper=0.;
1311    double y1lower=0.;
1312    double y2upper=0.;
1313    double y2lower=0.;
1314    double a=0.;
1315    double x3=0.;
1316    double y3=0.;
1317    double n3=0.;
1318    bool parallel=false;
1319    Matrix rot=Matrix(3);
1320    Vector n;
1321    std::vector<Point3D> v1=r1.get_vertices();
1322    std::vector<Point3D> v2=r2.get_vertices();
1323    switch (check_parallel(r1,r2))
1324    {
1325      case YZZX:
1326        rot.set(1,1,0);
1327        rot.set(1,2,-1);
1328        rot.set(1,3,0);
1329        rot.set(2,1,1);
1330        rot.set(2,2,0);
1331        rot.set(2,3,0);
1332        rot.set(3,1,0);
1333        rot.set(3,2,0);
1334        rot.set(3,3,1);
1335        v1[0]=MVM(rot,v1[0]);
```

```
1336        v1[1]=MVM(rot,v1[1]);
1337        v1[2]=MVM(rot,v1[2]);
1338        v1[3]=MVM(rot,v1[3]);
1339        v2[0]=MVM(rot,v2[0]);
1340        v2[1]=MVM(rot,v2[1]);
1341        v2[2]=MVM(rot,v2[2]);
1342        v2[3]=MVM(rot,v2[3]);
1343      case ZXYZ:
1344        rot.set(1,1,0);
1345        rot.set(1,2,0);
1346        rot.set(1,3,1);
1347        rot.set(2,1,0);
1348        rot.set(2,2,1);
1349        rot.set(2,3,0);
1350        rot.set(3,1,-1);
1351        rot.set(3,2,0);
1352        rot.set(3,3,0);
1353        v1[0]=MVM(rot,v1[0]);
1354        v1[1]=MVM(rot,v1[1]);
1355        v1[2]=MVM(rot,v1[2]);
1356        v1[3]=MVM(rot,v1[3]);
1357        v2[0]=MVM(rot,v2[0]);
1358        v2[1]=MVM(rot,v2[1]);
1359        v2[2]=MVM(rot,v2[2]);
1360        v2[3]=MVM(rot,v2[3]);
1361      case ZXXY:
1362        rot.set(1,1,1);
1363        rot.set(1,2,0);
1364        rot.set(1,3,0);
1365        rot.set(2,1,0);
1366        rot.set(2,2,0);
1367        rot.set(2,3,-1);
1368        rot.set(3,1,0);
1369        rot.set(3,2,1);
1370        rot.set(3,3,0);
1371        v1[0]=MVM(rot,v1[0]);
1372        v1[1]=MVM(rot,v1[1]);
1373        v1[2]=MVM(rot,v1[2]);
1374        v1[3]=MVM(rot,v1[3]);
1375        v2[0]=MVM(rot,v2[0]);
1376        v2[1]=MVM(rot,v2[1]);
1377        v2[2]=MVM(rot,v2[2]);
1378        v2[3]=MVM(rot,v2[3]);
1379      case XYZX:
1380        rot.set(1,1,0);
1381        rot.set(1,2,-1);
1382        rot.set(1,3,0);
1383        rot.set(2,1,1);
1384        rot.set(2,2,0);
1385        rot.set(2,3,0);
1386        rot.set(3,1,0);
1387        rot.set(3,2,0);
1388        rot.set(3,3,1);
1389        v1[0]=MVM(rot,v1[0]);
1390        v1[1]=MVM(rot,v1[1]);
1391        v1[2]=MVM(rot,v1[2]);
1392        v1[3]=MVM(rot,v1[3]);
1393        v2[0]=MVM(rot,v2[0]);
1394        v2[1]=MVM(rot,v2[1]);
1395        v2[2]=MVM(rot,v2[2]);
1396        v2[3]=MVM(rot,v2[3]);
1397      case XYYZ:
1398        rot.set(1,1,0);
```

```
1399        rot.set(1,2,0);
1400        rot.set(1,3,1);
1401        rot.set(2,1,0);
1402        rot.set(2,2,1);
1403        rot.set(2,3,0);
1404        rot.set(3,1,-1);
1405        rot.set(3,2,0);
1406        rot.set(3,3,0);
1407        v1[0]=MVM(rot,v1[0]);
1408        v1[1]=MVM(rot,v1[1]);
1409        v1[2]=MVM(rot,v1[2]);
1410        v1[3]=MVM(rot,v1[3]);
1411        v2[0]=MVM(rot,v2[0]);
1412        v2[1]=MVM(rot,v2[1]);
1413        v2[2]=MVM(rot,v2[2]);
1414        v2[3]=MVM(rot,v2[3]);
1415      case YZXY:
1416        n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
1417        normalize(n);
1418        n3=n.z();
1419        x1upper=v1[0].y();
1420        x1lower=v1[0].y();
1421        x2upper=v1[0].z();
1422        x2lower=v1[0].z();
1423        y1upper=v2[0].x();
1424        y1lower=v2[0].x();
1425        y2upper=v2[0].y();
1426        y2lower=v2[0].y();
1427        x3=v1[0].x();
1428        y3=v2[0].z();
1429        for(int i=1;i<4;++i)
1430        {
1431          if(v1[i].y()>x1upper)
1432            x1upper=v1[i].y();
1433          if(v1[i].y()<x1lower)
1434            x1lower=v1[i].y();
1435          if(v1[i].z()>x2upper)
1436            x2upper=v1[i].z();
1437          if(v1[i].z()<x2lower)
1438            x2lower=v1[i].z();
1439          if(v2[i].x()>y1upper)
1440            y1upper=v2[i].x();
1441          if(v2[i].x()<y1lower)
1442            y1lower=v2[i].x();
1443          if(v2[i].y()>y2upper)
1444            y2upper=v2[i].y();
1445          if(v2[i].y()<y2lower)
1446            y2lower=v2[i].y();
1447        }
1448        break;
1449      case YZYZ:
1450        rot.set(1,1,0);
1451        rot.set(1,2,-1);
1452        rot.set(1,3,0);
1453        rot.set(2,1,1);
1454        rot.set(2,2,0);
1455        rot.set(2,3,0);
1456        rot.set(3,1,0);
1457        rot.set(3,2,0);
1458        rot.set(3,3,1);
1459        v1[0]=MVM(rot,v1[0]);
1460        v1[1]=MVM(rot,v1[1]);
1461        v1[2]=MVM(rot,v1[2]);
```

```
1462        v1[3]=MVM(rot,v1[3]);
1463        v2[0]=MVM(rot,v2[0]);
1464        v2[1]=MVM(rot,v2[1]);
1465        v2[2]=MVM(rot,v2[2]);
1466        v2[3]=MVM(rot,v2[3]);
1467      case ZXZX:
1468        rot.set(1,1,1);
1469        rot.set(1,2,0);
1470        rot.set(1,3,0);
1471        rot.set(2,1,0);
1472        rot.set(2,2,0);
1473        rot.set(2,3,-1);
1474        rot.set(3,1,0);
1475        rot.set(3,2,1);
1476        rot.set(3,3,0);
1477        v1[0]=MVM(rot,v1[0]);
1478        v1[1]=MVM(rot,v1[1]);
1479        v1[2]=MVM(rot,v1[2]);
1480        v1[3]=MVM(rot,v1[3]);
1481        v2[0]=MVM(rot,v2[0]);
1482        v2[1]=MVM(rot,v2[1]);
1483        v2[2]=MVM(rot,v2[2]);
1484        v2[3]=MVM(rot,v2[3]);
1485      case XYXY:
1486        parallel=true;
1487        n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
1488        normalize(n);
1489        n3=n.z();
1490        x1upper=v1[0].x();
1491        x1lower=v1[0].x();
1492        x2upper=v1[0].y();
1493        x2lower=v1[0].y();
1494        y1upper=v2[0].x();
1495        y1lower=v2[0].x();
1496        y2upper=v2[0].y();
1497        y2lower=v2[0].y();
1498        x3=v1[0].z();
1499        y3=v2[0].z();
1500        for(int i=1;i<4;++i)
1501        {
1502          if(v1[i].x()>x1upper)
1503            x1upper=v1[i].x();
1504          if(v1[i].x()<x1lower)
1505            x1lower=v1[i].x();
1506          if(v1[i].y()>x2upper)
1507            x2upper=v1[i].y();
1508          if(v1[i].y()<x2lower)
1509            x2lower=v1[i].y();
1510          if(v2[i].x()>y1upper)
1511            y1upper=v2[i].x();
1512          if(v2[i].x()<y1lower)
1513            y1lower=v2[i].x();
1514          if(v2[i].y()>y2upper)
1515            y2upper=v2[i].y();
1516          if(v2[i].y()<y2lower)
1517            y2lower=v2[i].y();
1518        }
1519        break;
1520      default:
1521        break;
1522    }
1523    if (parallel)
1524    {
```

```
1525      result=eval_K(j,k,l,m,n3,-0.5,x1upper,x2upper,x3,y1upper,y2upper,y3,0,0,0)
1526          -eval_K(j,k,l,m,n3,-0.5,x1upper,x2upper,x3,y1upper,y2lower,y3,0,0,0)
1527          -eval_K(j,k,l,m,n3,-0.5,x1upper,x2upper,x3,y1lower,y2upper,y3,0,0,0)
1528          +eval_K(j,k,l,m,n3,-0.5,x1upper,x2upper,x3,y1lower,y2lower,y3,0,0,0)
1529          -eval_K(j,k,l,m,n3,-0.5,x1upper,x2lower,x3,y1upper,y2upper,y3,0,0,0)
1530          +eval_K(j,k,l,m,n3,-0.5,x1upper,x2lower,x3,y1upper,y2lower,y3,0,0,0)
1531          +eval_K(j,k,l,m,n3,-0.5,x1upper,x2lower,x3,y1lower,y2upper,y3,0,0,0)
1532          -eval_K(j,k,l,m,n3,-0.5,x1upper,x2lower,x3,y1lower,y2lower,y3,0,0,0)
1533          -eval_K(j,k,l,m,n3,-0.5,x1lower,x2upper,x3,y1upper,y2upper,y3,0,0,0)
1534          +eval_K(j,k,l,m,n3,-0.5,x1lower,x2upper,x3,y1upper,y2lower,y3,0,0,0)
1535          +eval_K(j,k,l,m,n3,-0.5,x1lower,x2upper,x3,y1lower,y2upper,y3,0,0,0)
1536          -eval_K(j,k,l,m,n3,-0.5,x1lower,x2upper,x3,y1lower,y2lower,y3,0,0,0)
1537          +eval_K(j,k,l,m,n3,-0.5,x1lower,x2lower,x3,y1upper,y2upper,y3,0,0,0)
1538          -eval_K(j,k,l,m,n3,-0.5,x1lower,x2lower,x3,y1upper,y2lower,y3,0,0,0)
1539          -eval_K(j,k,l,m,n3,-0.5,x1lower,x2lower,x3,y1lower,y2upper,y3,0,0,0)
1540          +eval_K(j,k,l,m,n3,-0.5,x1lower,x2lower,x3,y1lower,y2lower,y3,0,0,0);
1541    }
1542    else
1543    {
1544      result=eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2upper,y1upper,y2upper,y3,0,0,0)
1545          -eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2upper,y1upper,y2lower,y3,0,0,0)
1546          -eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2upper,y1lower,y2upper,y3,0,0,0)
1547          +eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2upper,y1lower,y2lower,y3,0,0,0)
1548          -eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2lower,y1upper,y2upper,y3,0,0,0)
1549          +eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2lower,y1upper,y2lower,y3,0,0,0)
1550          +eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2lower,y1lower,y2upper,y3,0,0,0)
1551          -eval_K2(j,k,l,m,n3,-0.5,x3,x1upper,x2lower,y1lower,y2lower,y3,0,0,0)
1552          -eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2upper,y1upper,y2upper,y3,0,0,0)
1553          +eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2upper,y1upper,y2lower,y3,0,0,0)
1554          +eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2upper,y1lower,y2upper,y3,0,0,0)
1555          -eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2upper,y1lower,y2lower,y3,0,0,0)
1556          +eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2lower,y1upper,y2upper,y3,0,0,0)
1557          -eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2lower,y1upper,y2lower,y3,0,0,0)
1558          -eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2lower,y1lower,y2upper,y3,0,0,0)
1559          +eval_K2(j,k,l,m,n3,-0.5,x3,x1lower,x2lower,y1lower,y2lower,y3,0,0,0);
1560    }
1561    return result/(4*PI);
1562  }
1563
1564  }
```

A.4.4 integral_operators.h

```
1  //***************************************************************************//
2  //File:                    integral_operators.h                              //
3  //This file declares and defines classes to represent the simplelayerpotential//
4  //and teh doublelayerpotential. The main routine in both cases is a method    //
5  //called build, which computes the entries using liblap3 from maiprogs        //
6  //***************************************************************************//
7  //Author: Samuel Ferraz-Leite                                                 //
8  //Date: 07.2007                                                               //
9  //Version 0.7 under development, beta-phase                                   //
10 //***************************************************************************//
11 #ifndef INTEGRAL_OPERATORS_H
12 #define INTEGRAL_OPERATORS_H
13
14 //Includes for the use of the hlib... copied from example_bem3d.c
15 extern "C"
16 {
17 #include "bem3d.h"
18 #include "cluster.h"
19 #include "h2conversion.h"
20 #include "h2virtual.h"
21 #include "h2apriori.h"
```

```
22 #include "h2arithmetics.h"
23 #include "hca.h"
24 #include "laplacebem.h"
25 #include "surfacebem.h"
26 #include "supermatrix.h"
27 #include "factorisations.h"
28 #include "hcoarsening.h"
29 #include "graphics.h"
30 #include "krylov.h"
31 #include "norm.h"
32 #include "maiprogs.h"
33 }
34
35 #include <assert.h>
36 #include <ctype.h>
37 #include <stdio.h>
38 #include <stdlib.h>
39 #include <math.h>
40 #include <sys/times.h>
41 #include <string.h>
42 #include <time.h>
43
44
45 //Includes from the bemlib for the adaptive mesh
46 #include "triangulation.h"
47 #include "triangulationio.h"
48 #include "quadrature.h"
49 #include "geometrics.h"
50
51 //Includes from the standardlibraries
52 #include <iostream>
53 #include<fstream>
54
55 #define ACA_KMAX 1e4
56 #define ACA_EPS 1e-6
57 #define ACA_STRATEGY HLIB_ACA_RELATIVE
58
59 using namespace triangulation;
60 using namespace geometrics;
61 using namespace quadrature;
62
63 namespace BEM {
64
65   //This very dirty structure needed to be created, since for the building
66   //of HH-Matrices the ACA-implementation of the HLib is used, which makes
67   //this necessity arise.
68   struct meshes_for_aca_{
69     Triangulation3D* row_mesh;
70     Triangulation3D* col_mesh;
71   };
72
73   typedef struct meshes_for_aca_ meshes_for_aca;
74
75   //-----------//
76   // Class SLP //
77   //-----------//
78   class SLP{
79     function_type ansatz_functions;
80     //These private routines are necessary since the use of HLib causes
81     //reordering of the degrees of freedom. FLBEM always provides ordering
82     //as is natural by the numbering of the mesh.
83     double* reorder(double* v,clustertree* ct);
84     double* disorder(double* v,clustertree* ct);
```

```
85      public:
86      ~SLP();//The destructor frees all data allocated by HLib.
87      supermatrix* A;//HH-Matrix = datafield (This may be a fullmatrix too)
88      clustertree* rct; //row-clustertree
89      clustertree* cct; //col-clustertree
90      supermatrix* build_preconditioner();
91
92      //Use this version of build, to create asymetrical mattrices
93      //or other special cases. If espilon=0 is given, a full matrix
94      //will be computed.
95      void build(Triangulation3D& from_mesh,Triangulation3D& to_mesh,
96                 function_type ansatz,double epsilon=ACA_EPS);
97
98      //Use this version of build like the first one, but for symmetric
99      //matrices only.
100     void build(Triangulation3D& mesh,function_type ansatz,double epsilon=ACA_EPS);
101     double* operator* (double* x);//Matrix-vector-multiplication
102     double* operator/ (double* b);//Solve linear System Ax=b by using x=A/b.
103   };
104
105   //-----------//
106   // Class DLP //
107   //-----------//
108
109   //The double-layer potential is still under development and so far only
110   //impolemented for the use of full-matrices. It is not yet fully tested.
111   class DLP_RHS{
112     public:
113       ~DLP_RHS();
114       Matrix A;
115       void build(Triangulation3D& from_mesh,Triangulation3D& to_mesh,function_type
              ansatz);
116       double* operator* (double* x);
117   };
118
119 }
120 #endif
```

A.4.5 integral_operators.cpp

```
1  //************************************************************************//
2  //File:                    integral_operators.cpp                       //
3  //This file implements all classes and routines specified in            //
4  //initegral_operators.h. Some additional routines are necessary, which need  //
5  //not to be visible are callable for the user.                           //
6  //************************************************************************//
7  //Author: Samuel Ferraz-Leite                                            //
8  //Date: 07.2007                                                          //
9  //Version 0.7 under development, beta-phase                              //
10 //************************************************************************//
11
12 //Include the header-file
13 #include "integral_operators.h"
14
15 namespace BEM {
16
17 //************************************************************************//
18 // ClusterFactory-Routines                                                //
19 //************************************************************************//
20 //This routine creates a clusterfactory structure, which is used in the
21 //process of building a clustertree by HLib. There are several versions
22 //avilable, one for each possible case of ansatz-functions.
23 clusterfactory* CreateClusterFactory(const Triangulation3D& mesh)
24 {
```

```
25    int n=mesh.size();
26    clusterfactory* factory=new_clusterfactory(n,n,3);
27    for(int i=0;i<n;++i)
28    {
29      std::vector<Point3D> v=mesh[i]->get_vertices();
30      double xmin=v[0].x();
31      double xmax=v[0].x();
32      double ymin=v[0].y();
33      double ymax=v[0].y();
34      double zmin=v[0].z();
35      double zmax=v[0].z();
36      for(int j=1;j<4;++j)
37      {
38        if(v[j].x()<xmin)
39          xmin=v[j].x();
40        if(v[j].x()>xmax)
41          xmax=v[j].x();
42        if(v[j].y()<ymin)
43          ymin=v[j].y();
44        if(v[j].y()>ymax)
45          ymax=v[j].y();
46        if(v[j].z()<zmin)
47          zmin=v[j].z();
48        if(v[j].z()>zmax)
49          zmax=v[j].z();
50      }
51      factory->x[i][0]=(xmin+xmax)/2.;
52      factory->x[i][1]=(ymin+ymax)/2.;
53      factory->x[i][2]=(zmin+zmax)/2.;
54
55      factory->smin[i][0]=xmin;
56      factory->smin[i][1]=ymin;
57      factory->smin[i][2]=zmin;
58      factory->smax[i][0]=xmax;
59      factory->smax[i][1]=ymax;
60      factory->smax[i][2]=zmax;
61    }
62    return factory;
63  }
64
65  clusterfactory* CreateClusterFactoryLinear(const Triangulation3D& mesh)
66  {
67    int n=4*mesh.size();
68    clusterfactory* factory=new_clusterfactory(n,n,3);
69    for(int i=0;i<mesh.size();++i)
70    {
71      std::vector<Point3D> v=mesh[i]->get_vertices();
72      double xmin=v[0].x();
73      double xmax=v[0].x();
74      double ymin=v[0].y();
75      double ymax=v[0].y();
76      double zmin=v[0].z();
77      double zmax=v[0].z();
78      for(int j=1;j<4;++j)
79      {
80        if(v[j].x()<xmin)
81          xmin=v[j].x();
82        if(v[j].x()>xmax)
83          xmax=v[j].x();
84        if(v[j].y()<ymin)
85          ymin=v[j].y();
86        if(v[j].y()>ymax)
87          ymax=v[j].y();
```

```
88        if(v[j].z()<zmin)
89          zmin=v[j].z();
90        if(v[j].z()>zmax)
91          zmax=v[j].z();
92      }
93      factory->x[4*i][0]=(xmin+xmax)/2.;
94      factory->x[4*i][1]=(ymin+ymax)/2.;
95      factory->x[4*i][2]=(zmin+zmax)/2.;
96      factory->x[4*i+1][0]=(xmin+xmax)/2.;
97      factory->x[4*i+1][1]=(ymin+ymax)/2.;
98      factory->x[4*i+1][2]=(zmin+zmax)/2.;
99      factory->x[4*i+2][0]=(xmin+xmax)/2.;
100     factory->x[4*i+2][1]=(ymin+ymax)/2.;
101     factory->x[4*i+2][2]=(zmin+zmax)/2.;
102     factory->x[4*i+3][0]=(xmin+xmax)/2.;
103     factory->x[4*i+3][1]=(ymin+ymax)/2.;
104     factory->x[4*i+3][2]=(zmin+zmax)/2.;
105
106     factory->smin[4*i][0]=xmin;
107     factory->smin[4*i][1]=ymin;
108     factory->smin[4*i][2]=zmin;
109     factory->smax[4*i][0]=xmax;
110     factory->smax[4*i][1]=ymax;
111     factory->smax[4*i][2]=zmax;
112     factory->smin[4*i+1][0]=xmin;
113     factory->smin[4*i+1][1]=ymin;
114     factory->smin[4*i+1][2]=zmin;
115     factory->smax[4*i+1][0]=xmax;
116     factory->smax[4*i+1][1]=ymax;
117     factory->smax[4*i+1][2]=zmax;
118     factory->smin[4*i+2][0]=xmin;
119     factory->smin[4*i+2][1]=ymin;
120     factory->smin[4*i+2][2]=zmin;
121     factory->smax[4*i+2][0]=xmax;
122     factory->smax[4*i+2][1]=ymax;
123     factory->smax[4*i+2][2]=zmax;
124     factory->smin[4*i+3][0]=xmin;
125     factory->smin[4*i+3][1]=ymin;
126     factory->smin[4*i+3][2]=zmin;
127     factory->smax[4*i+3][0]=xmax;
128     factory->smax[4*i+3][1]=ymax;
129     factory->smax[4*i+3][2]=zmax;
130   }
131   return factory;
132 }
133
134 //************************************************************************//
135 // Computatation of the entries                                           //
136 //************************************************************************//
137 //There are several function all starting with compute_entry_SLP which
138 //compute an entry A_ij in the stiffness matrix A given the indices and some
139 //additional information (i.e. the mesh and theclustertrees). The additional
140 //information named data hat to be declared void* to satisfy the function-pointer
141 //specification in the HLib for the ACA-algorithm.
142 double compute_entry_SLP_const(int i,int j,void* data)
143 {
144   bool is_parallel=false;
145   meshes_for_aca* meshdat=(meshes_for_aca*)data;
146   Triangulation3D* row_mesh=(Triangulation3D*) meshdat->row_mesh;
147   Triangulation3D* col_mesh=(Triangulation3D*) meshdat->col_mesh;
148   clustertree* rct=(clustertree*)row_mesh->data;
149   clustertree* cct=(clustertree*)col_mesh->data;
150   int I=rct->dof2idx[i];
```

```
151   int J=cct->dof2idx[j];
152
153   //Get elements
154   Shape* R1=(*row_mesh)[I];
155   Shape* R2=(*col_mesh)[J];
156
157   //get vertices
158   std::vector<Point3D> v1=R1->get_vertices();
159   std::vector<Point3D> v2=R2->get_vertices();
160
161   //Create rotational matrices. Since the antiderivatives only work for a
162   //specific geometrical alignement the given rectangles must be rotated
163   //to lie in the correct planes.
164   Matrix x_rot=Matrix(3);
165   x_rot.set(1,1,1);
166   x_rot.set(1,2,0);
167   x_rot.set(1,3,0);
168   x_rot.set(2,1,0);
169   x_rot.set(2,2,0);
170   x_rot.set(2,3,-1);
171   x_rot.set(3,1,0);
172   x_rot.set(3,2,1);
173   x_rot.set(3,3,0);
174   Matrix y_rot=Matrix(3);
175   y_rot.set(1,1,0);
176   y_rot.set(1,2,0);
177   y_rot.set(1,3,1);
178   y_rot.set(2,1,0);
179   y_rot.set(2,2,1);
180   y_rot.set(2,3,0);
181   y_rot.set(3,1,-1);
182   y_rot.set(3,2,0);
183   y_rot.set(3,3,0);
184   Matrix z_rot=Matrix(3);
185   z_rot.set(1,1,0);
186   z_rot.set(1,2,-1);
187   z_rot.set(1,3,0);
188   z_rot.set(2,1,1);
189   z_rot.set(2,2,0);
190   z_rot.set(2,3,0);
191   z_rot.set(3,1,0);
192   z_rot.set(3,2,0);
193   z_rot.set(3,3,1);
194
195   switch(check_parallel(*R1,*R2))
196   {
197   case XYXY:
198     is_parallel=true;
199     break;
200   case XYYZ:
201     for(int i=0;i<4;++i)
202     {
203       v1[i]=MVM(z_rot,v1[i]);
204       v2[i]=MVM(z_rot,v2[i]);
205     }
206     break;
207   case XYZX:
208     break;
209   case YZXY:
210     for(int i=0;i<4;++i)
211     {
212       v1[i]=MVM(y_rot,v1[i]);
213       v2[i]=MVM(y_rot,v2[i]);
```

```
214       }
215       for(int i=0;i<4;++i)
216       {
217         v1[i]=MVM(z_rot,v1[i]);
218         v2[i]=MVM(z_rot,v2[i]);
219       }
220       break;
221     case YZYZ:
222       is_parallel=true;
223       for(int i=0;i<4;++i)
224       {
225         v1[i]=MVM(y_rot,v1[i]);
226         v2[i]=MVM(y_rot,v2[i]);
227       }
228       break;
229     case YZZX:
230       for(int i=0;i<4;++i)
231       {
232         v1[i]=MVM(y_rot,v1[i]);
233         v2[i]=MVM(y_rot,v2[i]);
234       }
235       break;
236     case ZXXY:
237       for(int i=0;i<4;++i)
238       {
239         v1[i]=MVM(x_rot,v1[i]);
240         v2[i]=MVM(x_rot,v2[i]);
241       }
242       break;
243     case ZXYZ:
244       for(int i=0;i<4;++i)
245       {
246         v1[i]=MVM(x_rot,v1[i]);
247         v2[i]=MVM(x_rot,v2[i]);
248       }
249       for(int i=0;i<4;++i)
250       {
251         v1[i]=MVM(z_rot,v1[i]);
252         v2[i]=MVM(z_rot,v2[i]);
253       }
254       break;
255     case ZXZX:
256       is_parallel=true;
257       for(int i=0;i<4;++i)
258       {
259         v1[i]=MVM(x_rot,v1[i]);
260         v2[i]=MVM(x_rot,v2[i]);
261       }
262       break;
263     default:
264       break;
265     }
266     double* bx=(double*)malloc(3*sizeof(double));
267     double* dx1=(double*)malloc(3*sizeof(double));
268     double* dx2=(double*)malloc(3*sizeof(double));
269     double* nx=(double*)malloc(3*sizeof(double));
270     int tx=4;
271
272     double* by=(double*)malloc(3*sizeof(double));
273     double* dy1=(double*)malloc(3*sizeof(double));
274     double* dy2=(double*)malloc(3*sizeof(double));
275     double* ny=(double*)malloc(3*sizeof(double));
276     int ty=4;
```

```
277
278   int* pvx=(int*)malloc(2*sizeof(int));
279   pvx[0]=0;pvx[1]=0;
280   int* pvy=(int*)malloc(2*sizeof(int));
281   pvy[0]=0;pvy[1]=0;
282   int* pkx=(int*)malloc(2*sizeof(int));
283   pkx[0]=-1;pkx[1]=-1;
284   int* pky=(int*)malloc(2*sizeof(int));
285   pky[0]=-1;pky[1]=-1;
286   int* pwx=(int*)malloc(2*sizeof(int));
287   pwx[0]=-1;pwx[1]=-1;
288   int* pwy=(int*)malloc(2*sizeof(int));
289   pwy[0]=-1;pwy[1]=-1;
290
291   double* vklmn=(double*)malloc(10*10*10*10*sizeof(double));
292   double* kklmn=(double*)malloc(sizeof(double));
293   double* wklmn=(double*)malloc(sizeof(double));
294   vklmn[0]=0.;
295
296   if(is_parallel)
297   {
298     //Compute bx,dx1,dx2,etc...
299     double xmin=v1[0].x();
300     double xmax=v1[0].x();
301     double ymin=v1[0].y();
302     double ymax=v1[0].y();
303     double z=v1[0].z();
304
305     for(int i=1;i<4;++i)
306     {
307       if(xmin>v1[i].x())
308         xmin=v1[i].x();
309       if(xmax<v1[i].x())
310         xmax=v1[i].x();
311       if(ymin>v1[i].y())
312         ymin=v1[i].y();
313       if(ymax<v1[i].y())
314         ymax=v1[i].y();
315     }
316     bx[0]=xmin;
317     bx[1]=ymin;
318     bx[2]=z;
319
320     dx1[0]=xmax-xmin;
321     dx1[1]=0.;
322     dx1[2]=0.;
323     dx2[0]=0.;
324     dx2[1]=ymax-ymin;
325     dx2[2]=0.;
326
327     xmin=v2[0].x();
328     xmax=v2[0].x();
329     ymin=v2[0].y();
330     ymax=v2[0].y();
331     z=v2[0].z();
332
333     for(int i=1;i<4;++i)
334     {
335       if(xmin>v2[i].x())
336         xmin=v2[i].x();
337       if(xmax<v2[i].x())
338         xmax=v2[i].x();
339       if(ymin>v2[i].y())
```

```
340         ymin=v2[i].y();
341       if(ymax<v2[i].y())
342         ymax=v2[i].y();
343     }
344     by[0]=xmin;
345     by[1]=ymin;
346     by[2]=z;
347 
348     dy1[0]=xmax-xmin;
349     dy1[1]=0.;
350     dy1[2]=0.;
351     dy2[0]=0.;
352     dy2[1]=ymax-ymin;
353     dy2[2]=0.;
354   }
355   else
356   {
357     //Compute bx,dx1,dx2,etc...
358     double xmin=v1[0].x();
359     double xmax=v1[0].x();
360     double ymin=v1[0].y();
361     double ymax=v1[0].y();
362     double zmin=v1[0].z();
363     double zmax=v1[0].z();
364 
365     for(int i=1;i<4;++i)
366     {
367       if(xmin>v1[i].x())
368         xmin=v1[i].x();
369       if(xmax<v1[i].x())
370         xmax=v1[i].x();
371       if(ymin>v1[i].y())
372         ymin=v1[i].y();
373       if(ymax<v1[i].y())
374         ymax=v1[i].y();
375     }
376     bx[0]=xmin;
377     bx[1]=ymin;
378     bx[2]=zmin;
379 
380     dx1[0]=xmax-xmin;
381     dx1[1]=0.;
382     dx1[2]=0.;
383     dx2[0]=0.;
384     dx2[1]=ymax-ymin;
385     dx2[2]=0.;
386 
387     xmin=v2[0].x();
388     xmax=v2[0].x();
389     ymin=v2[0].y();
390     ymax=v2[0].y();
391     zmin=v2[0].z();
392     zmax=v2[0].z();
393 
394     for(int i=1;i<4;++i)
395     {
396       if(xmin>v2[i].x())
397         xmin=v2[i].x();
398       if(xmax<v2[i].x())
399         xmax=v2[i].x();
400       if(zmin>v2[i].z())
401         zmin=v2[i].z();
402       if(zmax<v2[i].z())
```

```
403           zmax=v2[i].z();
404       }
405       by[0]=xmin;
406       by[1]=ymin;
407       by[2]=zmin;
408
409       dy1[0]=xmax-xmin;
410       dy1[1]=0.;
411       dy1[2]=0.;
412       dy2[0]=0.;
413       dy2[1]=0.;
414       dy2[2]=zmax-zmin;
415     }
416
417     liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
418                          by,dy1,dy2,ny,&ty,
419                          pvx,pvy,pkx,pky,pwx,pwy,
420                          vklmn,kklmn,wklmn);
421
422     double result=vklmn[0]/2.;
423
424     free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
425     free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
426     free(vklmn);free(kklmn);free(wklmn);
427
428     return result;
429   }
430
431   double compute_entry_SLP_linear(int i,int j,void* data)
432   {
433     double factor=1.;
434
435     Matrix x_rot=Matrix(3);
436     x_rot.set(1,1,1);
437     x_rot.set(1,2,0);
438     x_rot.set(1,3,0);
439     x_rot.set(2,1,0);
440     x_rot.set(2,2,0);
441     x_rot.set(2,3,-1);
442     x_rot.set(3,1,0);
443     x_rot.set(3,2,1);
444     x_rot.set(3,3,0);
445     Matrix y_rot=Matrix(3);
446     y_rot.set(1,1,0);
447     y_rot.set(1,2,0);
448     y_rot.set(1,3,1);
449     y_rot.set(2,1,0);
450     y_rot.set(2,2,1);
451     y_rot.set(2,3,0);
452     y_rot.set(3,1,-1);
453     y_rot.set(3,2,0);
454     y_rot.set(3,3,0);
455     Matrix z_rot=Matrix(3);
456     z_rot.set(1,1,0);
457     z_rot.set(1,2,-1);
458     z_rot.set(1,3,0);
459     z_rot.set(2,1,1);
460     z_rot.set(2,2,0);
461     z_rot.set(2,3,0);
462     z_rot.set(3,1,0);
463     z_rot.set(3,2,0);
464     z_rot.set(3,3,1);
465
```

```
466   double* bx=(double*)malloc(3*sizeof(double));
467   double* dx1=(double*)malloc(3*sizeof(double));
468   double* dx2=(double*)malloc(3*sizeof(double));
469   double* nx=(double*)malloc(3*sizeof(double));
470   int tx=4;
471
472   double* by=(double*)malloc(3*sizeof(double));
473   double* dy1=(double*)malloc(3*sizeof(double));
474   double* dy2=(double*)malloc(3*sizeof(double));
475   double* ny=(double*)malloc(3*sizeof(double));
476   int ty=4;
477
478   int* pvx=(int*)malloc(2*sizeof(int));
479   pvx[0]=1;pvx[1]=1;
480   int* pvy=(int*)malloc(2*sizeof(int));
481   pvy[0]=1;pvy[1]=1;
482   int* pkx=(int*)malloc(2*sizeof(int));
483   pkx[0]=-1;pkx[1]=-1;
484   int* pky=(int*)malloc(2*sizeof(int));
485   pky[0]=-1;pky[1]=-1;
486   int* pwx=(int*)malloc(2*sizeof(int));
487   pwx[0]=-1;pwx[1]=-1;
488   int* pwy=(int*)malloc(2*sizeof(int));
489   pwy[0]=-1;pwy[1]=-1;
490
491   double* vklmn=(double*)malloc(2000*sizeof(double));
492   double* kklmn=(double*)malloc(sizeof(double));
493   double* wklmn=(double*)malloc(sizeof(double));
494   for(int k=0;k<2000;++k)
495     vklmn[k]=0.;
496
497   meshes_for_aca* meshdat=(meshes_for_aca*)data;
498   Triangulation3D* row_mesh=(Triangulation3D*) meshdat->row_mesh;
499   Triangulation3D* col_mesh=(Triangulation3D*) meshdat->col_mesh;
500   clustertree* rct=(clustertree*)row_mesh->data;
501   clustertree* cct=(clustertree*)col_mesh->data;
502   int I=(rct->dof2idx[i])/4;
503   int J=(cct->dof2idx[j])/4;
504
505   Shape* R1=(*row_mesh)[I];
506   Shape* R2=(*col_mesh)[J];
507
508
509   I=rct->dof2idx[i];
510   J=cct->dof2idx[j];
511   int psi1_xpos=0;
512   int psi1_ypos=0;
513   int psi2_xpos=0;
514   int psi2_ypos=0;
515   std::vector<Point3D> v1=R1->get_vertices();
516   std::vector<Point3D> v2=R2->get_vertices();
517   bool is_parallel=false;
518   switch (check_parallel(*R1,*R2))
519   {
520     case XYXY:
521       is_parallel=true;
522       break;
523     case XYYZ:
524       for(int i=0;i<4;++i)
525       {
526         v1[i]=MVM(z_rot,v1[i]);
527         v2[i]=MVM(z_rot,v2[i]);
528       }
```

```
529       break;
530     case XYZX:
531       break;
532     case YZXY:
533       for(int i=0;i<4;++i)
534       {
535         v1[i]=MVM(y_rot,v1[i]);
536         v2[i]=MVM(y_rot,v2[i]);
537       }
538       for(int i=0;i<4;++i)
539       {
540         v1[i]=MVM(z_rot,v1[i]);
541         v2[i]=MVM(z_rot,v2[i]);
542       }
543       break;
544     case YZYZ:
545       is_parallel=true;
546       for(int i=0;i<4;++i)
547       {
548         v1[i]=MVM(y_rot,v1[i]);
549         v2[i]=MVM(y_rot,v2[i]);
550       }
551       break;
552     case YZZX:
553       for(int i=0;i<4;++i)
554       {
555         v1[i]=MVM(y_rot,v1[i]);
556         v2[i]=MVM(y_rot,v2[i]);
557       }
558       break;
559     case ZXXY:
560       for(int i=0;i<4;++i)
561       {
562         v1[i]=MVM(x_rot,v1[i]);
563         v2[i]=MVM(x_rot,v2[i]);
564       }
565       break;
566     case ZXYZ:
567       for(int i=0;i<4;++i)
568       {
569         v1[i]=MVM(x_rot,v1[i]);
570         v2[i]=MVM(x_rot,v2[i]);
571       }
572       for(int i=0;i<4;++i)
573       {
574         v1[i]=MVM(z_rot,v1[i]);
575         v2[i]=MVM(z_rot,v2[i]);
576       }
577       break;
578     case ZXZX:
579       is_parallel=true;
580       for(int i=0;i<4;++i)
581       {
582         v1[i]=MVM(x_rot,v1[i]);
583         v2[i]=MVM(x_rot,v2[i]);
584       }
585       break;
586     default:
587       std::cout<<"Wrong␣Case!\n";
588       exit(1);
589       break;
590   }
591   double x1upper=0.;
```

```
592   double x1lower=0.;
593   double x2upper=0.;
594   double x2lower=0.;
595   double y1upper=0.;
596   double y1lower=0.;
597   double y2upper=0.;
598   double y2lower=0.;
599   if(is_parallel)
600   {
601     x1upper=v1[0].x();
602     x1lower=v1[0].x();
603     x2upper=v1[0].y();
604     x2lower=v1[0].y();
605     y1upper=v2[0].x();
606     y1lower=v2[0].x();
607     y2upper=v2[0].y();
608     y2lower=v2[0].y();
609       //check which basis functions are meant. xpos bzw. ypos denote
610       //weather the basis-function has its peak left (xpos=0) or right
611       //(xpos=1) and front (ypos=0) or back (ypos=1)
612     for(int i=1;i<4;++i)
613     {
614       if(v1[i].x()>x1upper)
615         x1upper=v1[i].x();
616       if(v1[i].x()<x1lower)
617         x1lower=v1[i].x();
618       if(v1[i].y()>x2upper)
619         x2upper=v1[i].y();
620       if(v1[i].y()<x2lower)
621         x2lower=v1[i].y();
622       if(v2[i].x()>y1upper)
623         y1upper=v2[i].x();
624       if(v2[i].x()<y1lower)
625         y1lower=v2[i].x();
626       if(v2[i].y()>y2upper)
627         y2upper=v2[i].y();
628       if(v2[i].y()<y2lower)
629         y2lower=v2[i].y();
630     }
631     switch (I%4)
632     {
633       case 0:
634         if(v1[0].x()==x1lower)
635           psi1_xpos=0;
636         else
637           psi1_xpos=1;
638         if(v1[0].y()==x2lower)
639           psi1_ypos=0;
640         else
641           psi1_ypos=1;
642         break;
643       case 1:
644         if(v1[1].x()==x1lower)
645           psi1_xpos=0;
646         else
647           psi1_xpos=1;
648         if(v1[1].y()==x2lower)
649           psi1_ypos=0;
650         else
651           psi1_ypos=1;
652         break;
653       case 2:
654         if(v1[2].x()==x1lower)
```

```
655           psi1_xpos=0;
656         else
657           psi1_xpos=1;
658         if(v1[2].y()==x2lower)
659           psi1_ypos=0;
660         else
661           psi1_ypos=1;
662         break;
663       case 3:
664         if(v1[3].x()==x1lower)
665           psi1_xpos=0;
666         else
667           psi1_xpos=1;
668         if(v1[3].y()==x2lower)
669           psi1_ypos=0;
670         else
671           psi1_ypos=1;
672         break;
673     }
674     switch(J%4)
675     {
676       case 0:
677         if(v2[0].x()==y1lower)
678           psi2_xpos=0;
679         else
680           psi2_xpos=1;
681         if(v2[0].y()==y2lower)
682           psi2_ypos=0;
683         else
684           psi2_ypos=1;
685         break;
686       case 1:
687         if(v2[1].x()==y1lower)
688           psi2_xpos=0;
689         else
690           psi2_xpos=1;
691         if(v2[1].y()==y2lower)
692           psi2_ypos=0;
693         else
694           psi2_ypos=1;
695         break;
696       case 2:
697         if(v2[2].x()==y1lower)
698           psi2_xpos=0;
699         else
700           psi2_xpos=1;
701         if(v2[2].y()==y2lower)
702           psi2_ypos=0;
703         else
704           psi2_ypos=1;
705         break;
706       case 3:
707         if(v2[3].x()==y1lower)
708           psi2_xpos=0;
709         else
710           psi2_xpos=1;
711         if(v2[3].y()==y2lower)
712           psi2_ypos=0;
713         else
714           psi2_ypos=1;
715         break;
716     }
717     bx[0]=x1lower;
```

```
718       bx[1]=x2lower;
719       bx[2]=v1[0].z();
720
721       dx1[0]=x1upper-x1lower;
722       dx1[1]=0.;
723       dx1[2]=0.;
724       dx2[0]=0.;
725       dx2[1]=x2upper-x2lower;
726       dx2[2]=0.;
727
728       by[0]=y1lower;
729       by[1]=y2lower;
730       by[2]=v2[0].z();
731
732       dy1[0]=y1upper-y1lower;
733       dy1[1]=0.;
734       dy1[2]=0.;
735       dy2[0]=0.;
736       dy2[1]=y2upper-y2lower;
737       dy2[2]=0.;
738
739       liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
740                            by,dy1,dy2,ny,&ty,
741                            pvx,pvy,pkx,pky,pwx,pwy,
742                            vklmn,kklmn,wklmn);
743     }
744     else
745     {
746       x1upper=v1[0].x();
747       x1lower=v1[0].x();
748       x2upper=v1[0].y();
749       x2lower=v1[0].y();
750       y1upper=v2[0].x();
751       y1lower=v2[0].x();
752       y2upper=v2[0].z();
753       y2lower=v2[0].z();
754       for(int i=1;i<4;++i)
755       {
756         if(v1[i].x()>x1upper)
757           x1upper=v1[i].x();
758         if(v1[i].x()<x1lower)
759           x1lower=v1[i].x();
760         if(v1[i].y()>x2upper)
761           x2upper=v1[i].y();
762         if(v1[i].y()<x2lower)
763           x2lower=v1[i].y();
764         if(v2[i].x()>y1upper)
765           y1upper=v2[i].x();
766         if(v2[i].x()<y1lower)
767           y1lower=v2[i].x();
768         if(v2[i].z()>y2upper)
769           y2upper=v2[i].z();
770         if(v2[i].z()<y2lower)
771           y2lower=v2[i].z();
772       }
773       switch (I%4)
774       {
775         case 0:
776           if(v1[0].x()==x1lower)
777             psi1_xpos=0;
778           else
779             psi1_xpos=1;
780           if(v1[0].y()==x2lower)
```

```
781           psi1_ypos=0;
782         else
783           psi1_ypos=1;
784         break;
785       case 1:
786         if(v1[1].x()==x1lower)
787           psi1_xpos=0;
788         else
789           psi1_xpos=1;
790         if(v1[1].y()==x2lower)
791           psi1_ypos=0;
792         else
793           psi1_ypos=1;
794         break;
795       case 2:
796         if(v1[2].x()==x1lower)
797           psi1_xpos=0;
798         else
799           psi1_xpos=1;
800         if(v1[2].y()==x2lower)
801           psi1_ypos=0;
802         else
803           psi1_ypos=1;
804         break;
805       case 3:
806         if(v1[3].x()==x1lower)
807           psi1_xpos=0;
808         else
809           psi1_xpos=1;
810         if(v1[3].y()==x2lower)
811           psi1_ypos=0;
812         else
813           psi1_ypos=1;
814         break;
815     }
816     switch(J%4)
817     {
818       case 0:
819         if(v2[0].x()==y1lower)
820           psi2_xpos=0;
821         else
822           psi2_xpos=1;
823         if(v2[0].z()==y2lower)
824           psi2_ypos=0;
825         else
826           psi2_ypos=1;
827         break;
828       case 1:
829         if(v2[1].x()==y1lower)
830           psi2_xpos=0;
831         else
832           psi2_xpos=1;
833         if(v2[1].z()==y2lower)
834           psi2_ypos=0;
835         else
836           psi2_ypos=1;
837         break;
838       case 2:
839         if(v2[2].x()==y1lower)
840           psi2_xpos=0;
841         else
842           psi2_xpos=1;
843         if(v2[2].z()==y2lower)
```

```
844           psi2_ypos=0;
845         else
846           psi2_ypos=1;
847         break;
848       case 3:
849         if(v2[3].x()==y1lower)
850           psi2_xpos=0;
851         else
852           psi2_xpos=1;
853         if(v2[3].z()==y2lower)
854           psi2_ypos=0;
855         else
856           psi2_ypos=1;
857         break;
858     }
859     bx[0]=x1lower;
860     bx[1]=x2lower;
861     bx[2]=v1[0].z();
862 
863     dx1[0]=x1upper-x1lower;
864     dx1[1]=0.;
865     dx1[2]=0.;
866     dx2[0]=0.;
867     dx2[1]=x2upper-x2lower;
868     dx2[2]=0.;
869 
870     by[0]=y1lower;
871     by[1]=v2[0].y();
872     by[2]=y2lower;
873 
874     dy1[0]=y1upper-y1lower;
875     dy1[1]=0.;
876     dy1[2]=0.;
877     dy2[0]=0.;
878     dy2[1]=0.;
879     dy2[2]=y2upper-y2lower;
880 
881     liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
882                          by,dy1,dy2,ny,&ty,
883                          pvx,pvy,pkx,pky,pwx,pwy,
884                          vklmn,kklmn,wklmn);
885   }
886 
887   double s1=vklmn[0]/2.;
888   double s2=(vklmn[0]+vklmn[121])/4.;
889   double s3=(vklmn[0]+vklmn[1331])/4.;
890   double s4=(vklmn[0]+vklmn[121]+vklmn[1331]+vklmn[1452])/8.;
891   double s5=(vklmn[0]+vklmn[1])/4.;
892   double s6=(vklmn[0]+vklmn[1]+vklmn[121]+vklmn[122])/8.;
893   double s7=(vklmn[0]+vklmn[1]+vklmn[1331]+vklmn[1332])/8.;
894   double s8=(vklmn[0]+vklmn[1]+vklmn[121]+vklmn[122]+vklmn[1331]+
895              vklmn[1332]+vklmn[1452]+vklmn[1453])/16.;
896   double s9=(vklmn[0]+vklmn[11])/4.;
897   double s10=(vklmn[0]+vklmn[11]+vklmn[121]+vklmn[132])/8.;
898   double s11=(vklmn[0]+vklmn[11]+vklmn[1331]+vklmn[1342])/8.;
899   double s12=(vklmn[0]+vklmn[11]+vklmn[121]+vklmn[132]+vklmn[1331]+
900               vklmn[1342]+vklmn[1452]+vklmn[1463])/16.;
901   double s13=(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12])/8.;
902   double s14=(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12]+vklmn[121]+vklmn[122]+
903               vklmn[132]+vklmn[133])/16.;
904   double s15=(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12]+vklmn[1331]+vklmn[1332]+
905               vklmn[1342]+vklmn[1343])/16.;
906   double s16=(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12]+vklmn[121]+vklmn[122]+
```

```
907                 vklmn[132]+vklmn[133]+
908                 vklmn[1331]+vklmn[1332]+vklmn[1342]+vklmn[1343]+vklmn[1452]+
909                 vklmn[1453]+vklmn[1463]+vklmn[1464])/32.;
910
911   double result=0.;
912
913   if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==0) //psiA and psiA'
914     result=s1-s2-s3+s4-s5+s6+s7-s8-s9+s10+s11-s12+s13-s14-s15+s16;
915
916   if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==0)//psiA and psiB'
917     result=s2-s4-s6+s8-s10+s12+s14-s16;
918
919   if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==1)//psiA and psiC'
920     result=s4-s8-s12+s16;
921
922   if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==1)//psiA and psiD'
923     result=s3-s7-s11+s15-s4+s8+s12-s16;
924
925   if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==0)//psiB and psiA'
926     result=s5-s6-s7+s8-s13+s14+s15-s16;
927
928   if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==0)//psiB and psiB'
929     result=s6-s8-s14+s16;
930
931   if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==1)//psiB and psiC'
932     result=s8-s16;
933
934   if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==1)//psiB and psiD'
935     result=s7-s8-s15+s16;
936
937   if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==0)//psiC and psiA'
938     result=s13-s14-s15+s16;
939
940   if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==0)//psiC and psiB'
941     result=s14-s16;
942
943   if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==1)//psiC and psiC'
944     result=s16;
945
946   if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==1)//psiC and psiD'
947     result=s15-s16;
948
949   if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==0)//psiD and psiA'
950     result=s9-s10-s11+s12-s13+s14+s15-s16;
951
952   if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==0)//psiD and pisB'
953     result=s10-s12-s14+s16;
954
955   if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==1)//psiD and psiC'
956     result=s12-s16;
957
958   if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==1)//psiD and psiD'
959     result=s11-s12-s15+s16;
960
961   if(result<0)
962     result=0.;
963
964   free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
965   free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
966   free(vklmn);free(kklmn);free(wklmn);
967
968   return result;
969 }
```

```
970
971 double compute_SLP_linconst(int i,int j,void* data)
972 {
973   Matrix x_rot=Matrix(3);
974   x_rot.set(1,1,1);
975   x_rot.set(1,2,0);
976   x_rot.set(1,3,0);
977   x_rot.set(2,1,0);
978   x_rot.set(2,2,0);
979   x_rot.set(2,3,-1);
980   x_rot.set(3,1,0);
981   x_rot.set(3,2,1);
982   x_rot.set(3,3,0);
983   Matrix y_rot=Matrix(3);
984   y_rot.set(1,1,0);
985   y_rot.set(1,2,0);
986   y_rot.set(1,3,1);
987   y_rot.set(2,1,0);
988   y_rot.set(2,2,1);
989   y_rot.set(2,3,0);
990   y_rot.set(3,1,-1);
991   y_rot.set(3,2,0);
992   y_rot.set(3,3,0);
993   Matrix z_rot=Matrix(3);
994   z_rot.set(1,1,0);
995   z_rot.set(1,2,-1);
996   z_rot.set(1,3,0);
997   z_rot.set(2,1,1);
998   z_rot.set(2,2,0);
999   z_rot.set(2,3,0);
1000  z_rot.set(3,1,0);
1001  z_rot.set(3,2,0);
1002  z_rot.set(3,3,1);
1003
1004  double* bx=(double*)malloc(3*sizeof(double));
1005  double* dx1=(double*)malloc(3*sizeof(double));
1006  double* dx2=(double*)malloc(3*sizeof(double));
1007  double* nx=(double*)malloc(3*sizeof(double));
1008  int tx=4;
1009
1010  double* by=(double*)malloc(3*sizeof(double));
1011  double* dy1=(double*)malloc(3*sizeof(double));
1012  double* dy2=(double*)malloc(3*sizeof(double));
1013  double* ny=(double*)malloc(3*sizeof(double));
1014  int ty=4;
1015
1016  int* pvx=(int*)malloc(2*sizeof(int));
1017  pvx[0]=1;pvx[1]=1;
1018  int* pvy=(int*)malloc(2*sizeof(int));
1019  pvy[0]=0;pvy[1]=0;
1020  int* pkx=(int*)malloc(2*sizeof(int));
1021  pkx[0]=-1;pkx[1]=-1;
1022  int* pky=(int*)malloc(2*sizeof(int));
1023  pky[0]=-1;pky[1]=-1;
1024  int* pwx=(int*)malloc(2*sizeof(int));
1025  pwx[0]=-1;pwx[1]=-1;
1026  int* pwy=(int*)malloc(2*sizeof(int));
1027  pwy[0]=-1;pwy[1]=-1;
1028
1029  double* vklmn=(double*)malloc(20*sizeof(double));
1030  double* kklmn=(double*)malloc(sizeof(double));
1031  double* wklmn=(double*)malloc(sizeof(double));
1032  for(int k=0;k<20;++k)
```

```
1033      vklmn[k]=0.;
1034
1035    double suspectvalue=0.;
1036    Matrix rot=Matrix(3);
1037    meshes_for_aca* meshdat=(meshes_for_aca*)data;
1038    Triangulation3D* row_mesh=(Triangulation3D*) meshdat->row_mesh;
1039    Triangulation3D* col_mesh=(Triangulation3D*) meshdat->col_mesh;
1040    clustertree* rct=(clustertree*)row_mesh->data;
1041    clustertree* cct=(clustertree*)col_mesh->data;
1042    int I=(rct->dof2idx[i])/4;
1043    int J=(cct->dof2idx[j]);
1044
1045    Shape* R1=(*row_mesh)[I];
1046    Shape* R2=(*col_mesh)[J];
1047    I=rct->dof2idx[i];
1048    int psi1_xpos=0;
1049    int psi1_ypos=0;
1050    std::vector<Point3D> v1=R1->get_vertices();
1051    std::vector<Point3D> v2=R2->get_vertices();
1052    bool is_parallel=false;
1053
1054    switch (check_parallel(*R1,*R2))
1055    {
1056      case XYXY:
1057        is_parallel=true;
1058        break;
1059      case XYYZ:
1060        for(int i=0;i<4;++i)
1061        {
1062          v1[i]=MVM(z_rot,v1[i]);
1063          v2[i]=MVM(z_rot,v2[i]);
1064        }
1065        break;
1066      case XYZX:
1067        break;
1068      case YZXY:
1069        for(int i=0;i<4;++i)
1070        {
1071          v1[i]=MVM(y_rot,v1[i]);
1072          v2[i]=MVM(y_rot,v2[i]);
1073        }
1074        for(int i=0;i<4;++i)
1075        {
1076          v1[i]=MVM(z_rot,v1[i]);
1077          v2[i]=MVM(z_rot,v2[i]);
1078        }
1079        break;
1080      case YZYZ:
1081        is_parallel=true;
1082        for(int i=0;i<4;++i)
1083        {
1084          v1[i]=MVM(y_rot,v1[i]);
1085          v2[i]=MVM(y_rot,v2[i]);
1086        }
1087        break;
1088      case YZZX:
1089        for(int i=0;i<4;++i)
1090        {
1091          v1[i]=MVM(y_rot,v1[i]);
1092          v2[i]=MVM(y_rot,v2[i]);
1093        }
1094        break;
1095      case ZXXY:
```

```
1096        for(int i=0;i<4;++i)
1097        {
1098          v1[i]=MVM(x_rot,v1[i]);
1099          v2[i]=MVM(x_rot,v2[i]);
1100        }
1101        break;
1102      case ZXYZ:
1103        for(int i=0;i<4;++i)
1104        {
1105          v1[i]=MVM(x_rot,v1[i]);
1106          v2[i]=MVM(x_rot,v2[i]);
1107        }
1108        for(int i=0;i<4;++i)
1109        {
1110          v1[i]=MVM(z_rot,v1[i]);
1111          v2[i]=MVM(z_rot,v2[i]);
1112        }
1113        break;
1114      case ZXZX:
1115        is_parallel=true;
1116        for(int i=0;i<4;++i)
1117        {
1118          v1[i]=MVM(x_rot,v1[i]);
1119          v2[i]=MVM(x_rot,v2[i]);
1120        }
1121        break;
1122      default:
1123        std::cout<<"Wrong Case!\n";
1124        exit(1);
1125        break;
1126    }
1127    double x1upper=0.;
1128    double x1lower=0.;
1129    double x2upper=0.;
1130    double x2lower=0.;
1131    double y1upper=0.;
1132    double y1lower=0.;
1133    double y2upper=0.;
1134    double y2lower=0.;
1135    if(is_parallel)
1136    {
1137      x1upper=v1[0].x();
1138      x1lower=v1[0].x();
1139      x2upper=v1[0].y();
1140      x2lower=v1[0].y();
1141      y1upper=v2[0].x();
1142      y1lower=v2[0].x();
1143      y2upper=v2[0].y();
1144      y2lower=v2[0].y();
1145        //check which basis functions are meant. xpos bzw. ypos denote
1146        //weather the basis-function has its peak left (xpos=0) or right
1147        //(xpos=1) and front (ypos=0) or back (ypos=1)
1148      for(int i=1;i<4;++i)
1149      {
1150        if(v1[i].x()>x1upper)
1151          x1upper=v1[i].x();
1152        if(v1[i].x()<x1lower)
1153          x1lower=v1[i].x();
1154        if(v1[i].y()>x2upper)
1155          x2upper=v1[i].y();
1156        if(v1[i].y()<x2lower)
1157          x2lower=v1[i].y();
1158        if(v2[i].x()>y1upper)
```

```
1159          y1upper=v2[i].x();
1160        if(v2[i].x()<y1lower)
1161          y1lower=v2[i].x();
1162        if(v2[i].y()>y2upper)
1163          y2upper=v2[i].y();
1164        if(v2[i].y()<y2lower)
1165          y2lower=v2[i].y();
1166      }
1167      switch (I%4)
1168      {
1169        case 0:
1170          if(v1[0].x()==x1lower)
1171            psi1_xpos=0;
1172          else
1173            psi1_xpos=1;
1174          if(v1[0].y()==x2lower)
1175            psi1_ypos=0;
1176          else
1177            psi1_ypos=1;
1178          break;
1179        case 1:
1180          if(v1[1].x()==x1lower)
1181            psi1_xpos=0;
1182          else
1183            psi1_xpos=1;
1184          if(v1[1].y()==x2lower)
1185            psi1_ypos=0;
1186          else
1187            psi1_ypos=1;
1188          break;
1189        case 2:
1190          if(v1[2].x()==x1lower)
1191            psi1_xpos=0;
1192          else
1193            psi1_xpos=1;
1194          if(v1[2].y()==x2lower)
1195            psi1_ypos=0;
1196          else
1197            psi1_ypos=1;
1198          break;
1199        case 3:
1200          if(v1[3].x()==x1lower)
1201            psi1_xpos=0;
1202          else
1203            psi1_xpos=1;
1204          if(v1[3].y()==x2lower)
1205            psi1_ypos=0;
1206          else
1207            psi1_ypos=1;
1208          break;
1209      }
1210      bx[0]=x1lower;
1211      bx[1]=x2lower;
1212      bx[2]=v1[0].z();
1213
1214      dx1[0]=x1upper-x1lower;
1215      dx1[1]=0.;
1216      dx1[2]=0.;
1217      dx2[0]=0.;
1218      dx2[1]=x2upper-x2lower;
1219      dx2[2]=0.;
1220
1221      by[0]=y1lower;
```

```
1222      by[1]=y2lower;
1223      by[2]=v2[0].z();
1224
1225      dy1[0]=y1upper-y1lower;
1226      dy1[1]=0.;
1227      dy1[2]=0.;
1228      dy2[0]=0.;
1229      dy2[1]=y2upper-y2lower;
1230      dy2[2]=0.;
1231
1232      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
1233                           by,dy1,dy2,ny,&ty,
1234                           pvx,pvy,pkx,pky,pwx,pwy,
1235                           vklmn,kklmn,wklmn);
1236    }
1237    else
1238    {
1239      x1upper=v1[0].x();
1240      x1lower=v1[0].x();
1241      x2upper=v1[0].y();
1242      x2lower=v1[0].y();
1243      y1upper=v2[0].x();
1244      y1lower=v2[0].x();
1245      y2upper=v2[0].z();
1246      y2lower=v2[0].z();
1247      for(int i=1;i<4;++i)
1248      {
1249        if(v1[i].x()>x1upper)
1250          x1upper=v1[i].x();
1251        if(v1[i].x()<x1lower)
1252          x1lower=v1[i].x();
1253        if(v1[i].y()>x2upper)
1254          x2upper=v1[i].y();
1255        if(v1[i].y()<x2lower)
1256          x2lower=v1[i].y();
1257        if(v2[i].x()>y1upper)
1258          y1upper=v2[i].x();
1259        if(v2[i].x()<y1lower)
1260          y1lower=v2[i].x();
1261        if(v2[i].z()>y2upper)
1262          y2upper=v2[i].z();
1263        if(v2[i].z()<y2lower)
1264          y2lower=v2[i].z();
1265      }
1266      switch (I%4)
1267      {
1268        case 0:
1269          if(v1[0].x()==x1lower)
1270            psi1_xpos=0;
1271          else
1272            psi1_xpos=1;
1273          if(v1[0].y()==x2lower)
1274            psi1_ypos=0;
1275          else
1276            psi1_ypos=1;
1277          break;
1278        case 1:
1279          if(v1[1].x()==x1lower)
1280            psi1_xpos=0;
1281          else
1282            psi1_xpos=1;
1283          if(v1[1].y()==x2lower)
1284            psi1_ypos=0;
```

```
1285          else
1286            psi1_ypos=1;
1287          break;
1288        case 2:
1289          if(v1[2].x()==x1lower)
1290            psi1_xpos=0;
1291          else
1292            psi1_xpos=1;
1293          if(v1[2].y()==x2lower)
1294            psi1_ypos=0;
1295          else
1296            psi1_ypos=1;
1297          break;
1298        case 3:
1299          if(v1[3].x()==x1lower)
1300            psi1_xpos=0;
1301          else
1302            psi1_xpos=1;
1303          if(v1[3].y()==x2lower)
1304            psi1_ypos=0;
1305          else
1306            psi1_ypos=1;
1307          break;
1308      }
1309      bx[0]=x1lower;
1310      bx[1]=x2lower;
1311      bx[2]=v1[0].z();
1312
1313      dx1[0]=x1upper-x1lower;
1314      dx1[1]=0.;
1315      dx1[2]=0.;
1316      dx2[0]=0.;
1317      dx2[1]=x2upper-x2lower;
1318      dx2[2]=0.;
1319
1320      by[0]=y1lower;
1321      by[1]=v2[0].y();
1322      by[2]=y2lower;
1323
1324      dy1[0]=y1upper-y1lower;
1325      dy1[1]=0.;
1326      dy1[2]=0.;
1327      dy2[0]=0.;
1328      dy2[1]=0.;
1329      dy2[2]=y2upper-y2lower;
1330
1331      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
1332                           by,dy1,dy2,ny,&ty,
1333                           pvx,pvy,pkx,pky,pwx,pwy,
1334                           vklmn,kklmn,wklmn);
1335
1336    }
1337    double result=0.;
1338    if(psi1_xpos==0 && psi1_ypos==0) //psiA
1339      result=vklmn[0]/2.-(vklmn[0]+vklmn[1])/4.-(vklmn[0]+vklmn[11])/4.+
1340            (vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12])/8.;
1341
1342    if(psi1_xpos==1 && psi1_ypos==0)//psiB
1343      result=(vklmn[0]+vklmn[1])/4.-(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12])/8.;
1344
1345    if(psi1_xpos==1 && psi1_ypos==1)//psiC
1346      result=(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12])/8.;
1347
```

```
1348    if(psi1_xpos==0 && psi1_ypos==1)//psiD
1349      result=(vklmn[0]+vklmn[11])/4.-(vklmn[0]+vklmn[1]+vklmn[11]+vklmn[12])/8.;
1350
1351    free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
1352    free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
1353    free(vklmn);free(kklmn);free(wklmn);
1354
1355    return result;
1356  }
1357
1358  //The functions whose name starts with compute_enty_DLP work exactly like the ones
1359  //to compute the SLP-entries.
1360  double compute_entry_DLP_const(Triangulation3D& row_mesh,Triangulation3D& col_mesh,
1361                                 int i,int j)
1362  {
1363    return integrate_rectangle_dlp(*(row_mesh[i]),*(col_mesh[j]),0,0,0,0);
1364  }
1365
1366  double compute_DLP_linconst(Triangulation3D& row_mesh,Triangulation3D& col_mesh,
1367                              int i,int j)
1368  {
1369    Matrix x_rot=Matrix(3);
1370    x_rot.set(1,1,1);
1371    x_rot.set(1,2,0);
1372    x_rot.set(1,3,0);
1373    x_rot.set(2,1,0);
1374    x_rot.set(2,2,0);
1375    x_rot.set(2,3,-1);
1376    x_rot.set(3,1,0);
1377    x_rot.set(3,2,1);
1378    x_rot.set(3,3,0);
1379    Matrix y_rot=Matrix(3);
1380    y_rot.set(1,1,0);
1381    y_rot.set(1,2,0);
1382    y_rot.set(1,3,1);
1383    y_rot.set(2,1,0);
1384    y_rot.set(2,2,1);
1385    y_rot.set(2,3,0);
1386    y_rot.set(3,1,-1);
1387    y_rot.set(3,2,0);
1388    y_rot.set(3,3,0);
1389    Matrix z_rot=Matrix(3);
1390    z_rot.set(1,1,0);
1391    z_rot.set(1,2,-1);
1392    z_rot.set(1,3,0);
1393    z_rot.set(2,1,1);
1394    z_rot.set(2,2,0);
1395    z_rot.set(2,3,0);
1396    z_rot.set(3,1,0);
1397    z_rot.set(3,2,0);
1398    z_rot.set(3,3,1);
1399
1400    double* bx=(double*)malloc(3*sizeof(double));
1401    double* dx1=(double*)malloc(3*sizeof(double));
1402    double* dx2=(double*)malloc(3*sizeof(double));
1403    double* nx=(double*)malloc(3*sizeof(double));
1404    int tx=4;
1405
1406    double* by=(double*)malloc(3*sizeof(double));
1407    double* dy1=(double*)malloc(3*sizeof(double));
1408    double* dy2=(double*)malloc(3*sizeof(double));
1409    double* ny=(double*)malloc(3*sizeof(double));
1410    int ty=4;
```

```
1411
1412    int* pvx=(int*)malloc(2*sizeof(int));
1413    pvx[0]=-1;pvx[1]=-1;
1414    int* pvy=(int*)malloc(2*sizeof(int));
1415    pvy[0]=-1;pvy[1]=-1;
1416    int* pkx=(int*)malloc(2*sizeof(int));
1417    pkx[0]=1;pkx[1]=1;
1418    int* pky=(int*)malloc(2*sizeof(int));
1419    pky[0]=0;pky[1]=0;
1420    int* pwx=(int*)malloc(2*sizeof(int));
1421    pwx[0]=-1;pwx[1]=-1;
1422    int* pwy=(int*)malloc(2*sizeof(int));
1423    pwy[0]=-1;pwy[1]=-1;
1424
1425    double* vklmn=(double*)malloc(sizeof(double));
1426    double* kklmn=(double*)malloc(20*sizeof(double));
1427    double* wklmn=(double*)malloc(sizeof(double));
1428    for(int k=0;k<20;++k)
1429      kklmn[k]=0.;
1430
1431    double suspectvalue=0.;
1432    Matrix rot=Matrix(3);
1433    int I=i/4;
1434    int J=j;
1435
1436    Shape* R1=row_mesh[I];
1437    Shape* R2=col_mesh[J];
1438
1439    if(I==J)
1440      return R1->gram_det()/8.;
1441    I=i;
1442    int psi1_xpos=0;
1443    int psi1_ypos=0;
1444    std::vector<Point3D> v1=R1->get_vertices();
1445    std::vector<Point3D> v2=R2->get_vertices();
1446    bool is_parallel=false;
1447
1448    switch (check_parallel(*R1,*R2))
1449    {
1450      case XYXY:
1451        is_parallel=true;
1452        break;
1453      case XYYZ:
1454        for(int i=0;i<4;++i)
1455        {
1456          v1[i]=MVM(z_rot,v1[i]);
1457          v2[i]=MVM(z_rot,v2[i]);
1458        }
1459        break;
1460      case XYZX:
1461        break;
1462      case YZXY:
1463        for(int i=0;i<4;++i)
1464        {
1465          v1[i]=MVM(y_rot,v1[i]);
1466          v2[i]=MVM(y_rot,v2[i]);
1467        }
1468        for(int i=0;i<4;++i)
1469        {
1470          v1[i]=MVM(z_rot,v1[i]);
1471          v2[i]=MVM(z_rot,v2[i]);
1472        }
1473        break;
```

```
1474      case YZYZ:
1475        is_parallel=true;
1476        for(int i=0;i<4;++i)
1477        {
1478          v1[i]=MVM(y_rot,v1[i]);
1479          v2[i]=MVM(y_rot,v2[i]);
1480        }
1481        break;
1482      case YZZX:
1483        for(int i=0;i<4;++i)
1484        {
1485          v1[i]=MVM(y_rot,v1[i]);
1486          v2[i]=MVM(y_rot,v2[i]);
1487        }
1488        break;
1489      case ZXXY:
1490        for(int i=0;i<4;++i)
1491        {
1492          v1[i]=MVM(x_rot,v1[i]);
1493          v2[i]=MVM(x_rot,v2[i]);
1494        }
1495        break;
1496      case ZXYZ:
1497        for(int i=0;i<4;++i)
1498        {
1499          v1[i]=MVM(x_rot,v1[i]);
1500          v2[i]=MVM(x_rot,v2[i]);
1501        }
1502        for(int i=0;i<4;++i)
1503        {
1504          v1[i]=MVM(z_rot,v1[i]);
1505          v2[i]=MVM(z_rot,v2[i]);
1506        }
1507        break;
1508      case ZXZX:
1509        is_parallel=true;
1510        for(int i=0;i<4;++i)
1511        {
1512          v1[i]=MVM(x_rot,v1[i]);
1513          v2[i]=MVM(x_rot,v2[i]);
1514        }
1515        break;
1516      default:
1517        std::cout<<"Wrong␣Case!\n";
1518        exit(1);
1519        break;
1520    }
1521    double x1upper=0.;
1522    double x1lower=0.;
1523    double x2upper=0.;
1524    double x2lower=0.;
1525    double y1upper=0.;
1526    double y1lower=0.;
1527    double y2upper=0.;
1528    double y2lower=0.;
1529    if(is_parallel)
1530    {
1531      x1upper=v1[0].x();
1532      x1lower=v1[0].x();
1533      x2upper=v1[0].y();
1534      x2lower=v1[0].y();
1535      y1upper=v2[0].x();
1536      y1lower=v2[0].x();
```

```
1537      y2upper=v2[0].y();
1538      y2lower=v2[0].y();
1539        //check which basis functions are meant. xpos bzw. ypos denote
1540        //weather the basis-function has its peak left (xpos=0) or right
1541        //(xpos=1) and front (ypos=0) or back (ypos=1)
1542      for(int i=1;i<4;++i)
1543      {
1544        if(v1[i].x()>x1upper)
1545          x1upper=v1[i].x();
1546        if(v1[i].x()<x1lower)
1547          x1lower=v1[i].x();
1548        if(v1[i].y()>x2upper)
1549          x2upper=v1[i].y();
1550        if(v1[i].y()<x2lower)
1551          x2lower=v1[i].y();
1552        if(v2[i].x()>y1upper)
1553          y1upper=v2[i].x();
1554        if(v2[i].x()<y1lower)
1555          y1lower=v2[i].x();
1556        if(v2[i].y()>y2upper)
1557          y2upper=v2[i].y();
1558        if(v2[i].y()<y2lower)
1559          y2lower=v2[i].y();
1560      }
1561      switch (I%4)
1562      {
1563        case 0:
1564          if(v1[0].x()==x1lower)
1565            psi1_xpos=0;
1566          else
1567            psi1_xpos=1;
1568          if(v1[0].y()==x2lower)
1569            psi1_ypos=0;
1570          else
1571            psi1_ypos=1;
1572          break;
1573        case 1:
1574          if(v1[1].x()==x1lower)
1575            psi1_xpos=0;
1576          else
1577            psi1_xpos=1;
1578          if(v1[1].y()==x2lower)
1579            psi1_ypos=0;
1580          else
1581            psi1_ypos=1;
1582          break;
1583        case 2:
1584          if(v1[2].x()==x1lower)
1585            psi1_xpos=0;
1586          else
1587            psi1_xpos=1;
1588          if(v1[2].y()==x2lower)
1589            psi1_ypos=0;
1590          else
1591            psi1_ypos=1;
1592          break;
1593        case 3:
1594          if(v1[3].x()==x1lower)
1595            psi1_xpos=0;
1596          else
1597            psi1_xpos=1;
1598          if(v1[3].y()==x2lower)
1599            psi1_ypos=0;
```

```
1600          else
1601            psi1_ypos=1;
1602          break;
1603      }
1604      bx[0]=x1lower;
1605      bx[1]=x2lower;
1606      bx[2]=v1[0].z();
1607
1608      dx1[0]=x1upper-x1lower;
1609      dx1[1]=0.;
1610      dx1[2]=0.;
1611      dx2[0]=0.;
1612      dx2[1]=x2upper-x2lower;
1613      dx2[2]=0.;
1614
1615      by[0]=y1lower;
1616      by[1]=y2lower;
1617      by[2]=v2[0].z();
1618
1619      dy1[0]=y1upper-y1lower;
1620      dy1[1]=0.;
1621      dy1[2]=0.;
1622      dy2[0]=0.;
1623      dy2[1]=y2upper-y2lower;
1624      dy2[2]=0.;
1625
1626      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
1627      normalize(n);
1628      nx[0]=n.x();
1629      nx[1]=n.y();
1630      nx[2]=n.z();
1631
1632      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
1633      normalize(n);
1634      ny[0]=n.x();
1635      ny[1]=n.y();
1636      ny[2]=n.z();
1637
1638      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
1639                           by,dy1,dy2,ny,&ty,
1640                           pvx,pvy,pkx,pky,pwx,pwy,
1641                           vklmn,kklmn,wklmn);
1642    }
1643    else
1644    {
1645      x1upper=v1[0].x();
1646      x1lower=v1[0].x();
1647      x2upper=v1[0].y();
1648      x2lower=v1[0].y();
1649      y1upper=v2[0].x();
1650      y1lower=v2[0].x();
1651      y2upper=v2[0].z();
1652      y2lower=v2[0].z();
1653      for(int i=1;i<4;++i)
1654      {
1655        if(v1[i].x()>x1upper)
1656          x1upper=v1[i].x();
1657        if(v1[i].x()<x1lower)
1658          x1lower=v1[i].x();
1659        if(v1[i].y()>x2upper)
1660          x2upper=v1[i].y();
1661        if(v1[i].y()<x2lower)
1662          x2lower=v1[i].y();
```

```
1663        if(v2[i].x()>y1upper)
1664          y1upper=v2[i].x();
1665        if(v2[i].x()<y1lower)
1666          y1lower=v2[i].x();
1667        if(v2[i].z()>y2upper)
1668          y2upper=v2[i].z();
1669        if(v2[i].z()<y2lower)
1670          y2lower=v2[i].z();
1671      }
1672      switch (I%4)
1673      {
1674        case 0:
1675          if(v1[0].x()==x1lower)
1676            psi1_xpos=0;
1677          else
1678            psi1_xpos=1;
1679          if(v1[0].y()==x2lower)
1680            psi1_ypos=0;
1681          else
1682            psi1_ypos=1;
1683          break;
1684        case 1:
1685          if(v1[1].x()==x1lower)
1686            psi1_xpos=0;
1687          else
1688            psi1_xpos=1;
1689          if(v1[1].y()==x2lower)
1690            psi1_ypos=0;
1691          else
1692            psi1_ypos=1;
1693          break;
1694        case 2:
1695          if(v1[2].x()==x1lower)
1696            psi1_xpos=0;
1697          else
1698            psi1_xpos=1;
1699          if(v1[2].y()==x2lower)
1700            psi1_ypos=0;
1701          else
1702            psi1_ypos=1;
1703          break;
1704        case 3:
1705          if(v1[3].x()==x1lower)
1706            psi1_xpos=0;
1707          else
1708            psi1_xpos=1;
1709          if(v1[3].y()==x2lower)
1710            psi1_ypos=0;
1711          else
1712            psi1_ypos=1;
1713          break;
1714      }
1715      bx[0]=x1lower;
1716      bx[1]=x2lower;
1717      bx[2]=v1[0].z();
1718
1719      dx1[0]=x1upper-x1lower;
1720      dx1[1]=0.;
1721      dx1[2]=0.;
1722      dx2[0]=0.;
1723      dx2[1]=x2upper-x2lower;
1724      dx2[2]=0.;
1725
```

```
1726      by[0]=y1lower;
1727      by[1]=v2[0].y();
1728      by[2]=y2lower;
1729
1730      dy1[0]=y1upper-y1lower;
1731      dy1[1]=0.;
1732      dy1[2]=0.;
1733      dy2[0]=0.;
1734      dy2[1]=0.;
1735      dy2[2]=y2upper-y2lower;
1736
1737      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
1738      normalize(n);
1739      nx[0]=0.;
1740      nx[1]=0.;
1741      nx[2]=n.z();
1742
1743      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
1744      normalize(n);
1745      ny[0]=0.;
1746      ny[1]=-n.y();
1747      ny[2]=0.;
1748
1749      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
1750                           by,dy1,dy2,ny,&ty,
1751                           pvx,pvy,pkx,pky,pwx,pwy,
1752                           vklmn,kklmn,wklmn);
1753
1754    }
1755    double result=0.;
1756    if(psi1_xpos==0 && psi1_ypos==0) //psiA
1757      result=kklmn[0]/2.-(kklmn[0]+kklmn[1])/4.-(kklmn[0]+kklmn[11])/4.+
1758            (kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12])/8.;
1759
1760    if(psi1_xpos==1 && psi1_ypos==0)//psiB
1761      result=(kklmn[0]+kklmn[1])/4.-(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12])/8.;
1762
1763    if(psi1_xpos==1 && psi1_ypos==1)//psiC
1764      result=(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12])/8.;
1765
1766    if(psi1_xpos==0 && psi1_ypos==1)//psiD
1767      result=(kklmn[0]+kklmn[11])/4.-(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12])/8.;
1768
1769    free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
1770    free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
1771    free(vklmn);free(kklmn);free(wklmn);
1772
1773    return result;
1774  }
1775
1776  double compute_DLP_RHS_constlin(Triangulation3D& row_mesh, Triangulation3D& col_mesh
       ,
1777                                  int i,int j)
1778  {
1779    Matrix x_rot=Matrix(3);
1780    x_rot.set(1,1,1);
1781    x_rot.set(1,2,0);
1782    x_rot.set(1,3,0);
1783    x_rot.set(2,1,0);
1784    x_rot.set(2,2,0);
1785    x_rot.set(2,3,-1);
1786    x_rot.set(3,1,0);
1787    x_rot.set(3,2,1);
```

```
1788    x_rot.set(3,3,0);
1789    Matrix y_rot=Matrix(3);
1790    y_rot.set(1,1,0);
1791    y_rot.set(1,2,0);
1792    y_rot.set(1,3,1);
1793    y_rot.set(2,1,0);
1794    y_rot.set(2,2,1);
1795    y_rot.set(2,3,0);
1796    y_rot.set(3,1,-1);
1797    y_rot.set(3,2,0);
1798    y_rot.set(3,3,0);
1799    Matrix z_rot=Matrix(3);
1800    z_rot.set(1,1,0);
1801    z_rot.set(1,2,-1);
1802    z_rot.set(1,3,0);
1803    z_rot.set(2,1,1);
1804    z_rot.set(2,2,0);
1805    z_rot.set(2,3,0);
1806    z_rot.set(3,1,0);
1807    z_rot.set(3,2,0);
1808    z_rot.set(3,3,1);
1809
1810    double* bx=(double*)malloc(3*sizeof(double));
1811    double* dx1=(double*)malloc(3*sizeof(double));
1812    double* dx2=(double*)malloc(3*sizeof(double));
1813    double* nx=(double*)malloc(3*sizeof(double));
1814    int tx=4;
1815
1816    double* by=(double*)malloc(3*sizeof(double));
1817    double* dy1=(double*)malloc(3*sizeof(double));
1818    double* dy2=(double*)malloc(3*sizeof(double));
1819    double* ny=(double*)malloc(3*sizeof(double));
1820    int ty=4;
1821
1822    int* pvx=(int*)malloc(2*sizeof(int));
1823    pvx[0]=-1;pvx[1]=-1;
1824    int* pvy=(int*)malloc(2*sizeof(int));
1825    pvy[0]=-1;pvy[1]=-1;
1826    int* pkx=(int*)malloc(2*sizeof(int));
1827    pkx[0]=0;pkx[1]=0;
1828    int* pky=(int*)malloc(2*sizeof(int));
1829    pky[0]=1;pky[1]=1;
1830    int* pwx=(int*)malloc(2*sizeof(int));
1831    pwx[0]=-1;pwx[1]=-1;
1832    int* pwy=(int*)malloc(2*sizeof(int));
1833    pwy[0]=-1;pwy[1]=-1;
1834
1835    double* vklmn=(double*)malloc(sizeof(double));
1836    double* kklmn=(double*)malloc(1500*sizeof(double));
1837    double* wklmn=(double*)malloc(sizeof(double));
1838    for(int k=0;k<1500;++k)
1839      kklmn[k]=0.;
1840
1841    double suspectvalue=0.;
1842    Matrix rot=Matrix(3);
1843    int I=i;
1844    int J=j/4;
1845    Shape* R1=row_mesh[I];
1846    Shape* R2=col_mesh[J];
1847
1848    if(I==J)
1849      return 1./8.*R1->gram_det();
1850
```

```
1851    J=j;
1852    int psi_y1pos=0;
1853    int psi_y2pos=0;
1854    std::vector<Point3D> v1=R1->get_vertices();
1855    std::vector<Point3D> v2=R2->get_vertices();
1856    bool is_parallel=false;
1857
1858    switch (check_parallel(*R1,*R2))
1859    {
1860      case XYXY:
1861        is_parallel=true;
1862        break;
1863      case XYYZ:
1864        for(int i=0;i<4;++i)
1865        {
1866          v1[i]=MVM(z_rot,v1[i]);
1867          v2[i]=MVM(z_rot,v2[i]);
1868        }
1869        break;
1870      case XYZX:
1871        break;
1872      case YZXY:
1873        for(int i=0;i<4;++i)
1874        {
1875          v1[i]=MVM(y_rot,v1[i]);
1876          v2[i]=MVM(y_rot,v2[i]);
1877        }
1878        for(int i=0;i<4;++i)
1879        {
1880          v1[i]=MVM(z_rot,v1[i]);
1881          v2[i]=MVM(z_rot,v2[i]);
1882        }
1883        break;
1884      case YZYZ:
1885        is_parallel=true;
1886        for(int i=0;i<4;++i)
1887        {
1888          v1[i]=MVM(y_rot,v1[i]);
1889          v2[i]=MVM(y_rot,v2[i]);
1890        }
1891        break;
1892      case YZZX:
1893        for(int i=0;i<4;++i)
1894        {
1895          v1[i]=MVM(y_rot,v1[i]);
1896          v2[i]=MVM(y_rot,v2[i]);
1897        }
1898        break;
1899      case ZXXY:
1900        for(int i=0;i<4;++i)
1901        {
1902          v1[i]=MVM(x_rot,v1[i]);
1903          v2[i]=MVM(x_rot,v2[i]);
1904        }
1905        break;
1906      case ZXYZ:
1907        for(int i=0;i<4;++i)
1908        {
1909          v1[i]=MVM(x_rot,v1[i]);
1910          v2[i]=MVM(x_rot,v2[i]);
1911        }
1912        for(int i=0;i<4;++i)
1913        {
```

```
1914          v1[i]=MVM(z_rot,v1[i]);
1915          v2[i]=MVM(z_rot,v2[i]);
1916        }
1917        break;
1918      case ZXZX:
1919        is_parallel=true;
1920        for(int i=0;i<4;++i)
1921        {
1922          v1[i]=MVM(x_rot,v1[i]);
1923          v2[i]=MVM(x_rot,v2[i]);
1924        }
1925        break;
1926      default:
1927        std::cout<<"Wrong Case!\n";
1928        exit(1);
1929        break;
1930    }
1931    double x1upper=0.;
1932    double x1lower=0.;
1933    double x2upper=0.;
1934    double x2lower=0.;
1935    double y1upper=0.;
1936    double y1lower=0.;
1937    double y2upper=0.;
1938    double y2lower=0.;
1939    if(is_parallel)
1940    {
1941      x1upper=v1[0].x();
1942      x1lower=v1[0].x();
1943      x2upper=v1[0].y();
1944      x2lower=v1[0].y();
1945      y1upper=v2[0].x();
1946      y1lower=v2[0].x();
1947      y2upper=v2[0].y();
1948      y2lower=v2[0].y();
1949        //check which basis functions are meant. xpos bzw. ypos denote
1950        //weather the basis-function has its peak left (xpos=0) or right
1951        //(xpos=1) and front (ypos=0) or back (ypos=1)
1952      for(int i=1;i<4;++i)
1953      {
1954        if(v1[i].x()>x1upper)
1955          x1upper=v1[i].x();
1956        if(v1[i].x()<x1lower)
1957          x1lower=v1[i].x();
1958        if(v1[i].y()>x2upper)
1959          x2upper=v1[i].y();
1960        if(v1[i].y()<x2lower)
1961          x2lower=v1[i].y();
1962        if(v2[i].x()>y1upper)
1963          y1upper=v2[i].x();
1964        if(v2[i].x()<y1lower)
1965          y1lower=v2[i].x();
1966        if(v2[i].y()>y2upper)
1967          y2upper=v2[i].y();
1968        if(v2[i].y()<y2lower)
1969          y2lower=v2[i].y();
1970      }
1971      switch (J%4)
1972      {
1973        case 0:
1974          if(v2[0].x()==y1lower)
1975            psi_y1pos=0;
1976          else
```

```
1977            psi_y1pos=1;
1978          if(v2[0].y()==y2lower)
1979            psi_y2pos=0;
1980          else
1981            psi_y2pos=1;
1982          break;
1983        case 1:
1984          if(v2[1].x()==y1lower)
1985            psi_y1pos=0;
1986          else
1987            psi_y1pos=1;
1988          if(v2[1].y()==y2lower)
1989            psi_y2pos=0;
1990          else
1991            psi_y2pos=1;
1992          break;
1993        case 2:
1994          if(v2[2].x()==y1lower)
1995            psi_y1pos=0;
1996          else
1997            psi_y1pos=1;
1998          if(v2[2].y()==y2lower)
1999            psi_y2pos=0;
2000          else
2001            psi_y2pos=1;
2002          break;
2003        case 3:
2004          if(v2[3].x()==y1lower)
2005            psi_y1pos=0;
2006          else
2007            psi_y1pos=1;
2008          if(v2[3].y()==y2lower)
2009            psi_y2pos=0;
2010          else
2011            psi_y2pos=1;
2012          break;
2013      }
2014      bx[0]=x1lower;
2015      bx[1]=x2lower;
2016      bx[2]=v1[0].z();
2017
2018      dx1[0]=x1upper-x1lower;
2019      dx1[1]=0.;
2020      dx1[2]=0.;
2021      dx2[0]=0.;
2022      dx2[1]=x2upper-x2lower;
2023      dx2[2]=0.;
2024
2025      by[0]=y1lower;
2026      by[1]=y2lower;
2027      by[2]=v2[0].z();
2028
2029      dy1[0]=y1upper-y1lower;
2030      dy1[1]=0.;
2031      dy1[2]=0.;
2032      dy2[0]=0.;
2033      dy2[1]=y2upper-y2lower;
2034      dy2[2]=0.;
2035
2036      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
2037      normalize(n);
2038      nx[0]=n.x();
2039      nx[1]=n.y();
```

```
2040      nx[2]=n.z();
2041
2042      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
2043      normalize(n);
2044      ny[0]=n.x();
2045      ny[1]=n.y();
2046      ny[2]=n.z();
2047
2048      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
2049                           by,dy1,dy2,ny,&ty,
2050                           pvx,pvy,pkx,pky,pwx,pwy,
2051                           vklmn,kklmn,wklmn);
2052    }
2053    else
2054    {
2055      x1upper=v1[0].x();
2056      x1lower=v1[0].x();
2057      x2upper=v1[0].y();
2058      x2lower=v1[0].y();
2059      y1upper=v2[0].x();
2060      y1lower=v2[0].x();
2061      y2upper=v2[0].z();
2062      y2lower=v2[0].z();
2063      for(int i=1;i<4;++i)
2064      {
2065        if(v1[i].x()>x1upper)
2066          x1upper=v1[i].x();
2067        if(v1[i].x()<x1lower)
2068          x1lower=v1[i].x();
2069        if(v1[i].y()>x2upper)
2070          x2upper=v1[i].y();
2071        if(v1[i].y()<x2lower)
2072          x2lower=v1[i].y();
2073        if(v2[i].x()>y1upper)
2074          y1upper=v2[i].x();
2075        if(v2[i].x()<y1lower)
2076          y1lower=v2[i].x();
2077        if(v2[i].z()>y2upper)
2078          y2upper=v2[i].z();
2079        if(v2[i].z()<y2lower)
2080          y2lower=v2[i].z();
2081      }
2082      switch (J%4)
2083      {
2084        case 0:
2085          if(v2[0].x()==y1lower)
2086            psi_y1pos=0;
2087          else
2088            psi_y1pos=1;
2089          if(v2[0].z()==y2lower)
2090            psi_y2pos=0;
2091          else
2092            psi_y2pos=1;
2093          break;
2094        case 1:
2095          if(v2[1].x()==y1lower)
2096            psi_y1pos=0;
2097          else
2098            psi_y1pos=1;
2099          if(v2[1].z()==y2lower)
2100            psi_y2pos=0;
2101          else
2102            psi_y2pos=1;
```

```
2103          break;
2104        case 2:
2105          if(v2[2].x()==y1lower)
2106            psi_y1pos=0;
2107          else
2108            psi_y1pos=1;
2109          if(v2[2].z()==y2lower)
2110            psi_y2pos=0;
2111          else
2112            psi_y2pos=1;
2113          break;
2114        case 3:
2115          if(v2[3].x()==y1lower)
2116            psi_y1pos=0;
2117          else
2118            psi_y1pos=1;
2119          if(v2[3].z()==y2lower)
2120            psi_y2pos=0;
2121          else
2122            psi_y2pos=1;
2123          break;
2124      }
2125      bx[0]=x1lower;
2126      bx[1]=x2lower;
2127      bx[2]=v1[0].z();
2128
2129      dx1[0]=x1upper-x1lower;
2130      dx1[1]=0.;
2131      dx1[2]=0.;
2132      dx2[0]=0.;
2133      dx2[1]=x2upper-x2lower;
2134      dx2[2]=0.;
2135
2136      by[0]=y1lower;
2137      by[1]=v2[0].y();
2138      by[2]=y2lower;
2139
2140      dy1[0]=y1upper-y1lower;
2141      dy1[1]=0.;
2142      dy1[2]=0.;
2143      dy2[0]=0.;
2144      dy2[1]=0.;
2145      dy2[2]=y2upper-y2lower;
2146
2147      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
2148      normalize(n);
2149      nx[0]=n.x();
2150      nx[1]=n.y();
2151      nx[2]=n.z();
2152
2153      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
2154      normalize(n);
2155      ny[0]=0.;
2156      ny[1]=-n.y();
2157      ny[2]=0.;
2158
2159      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
2160                           by,dy1,dy2,ny,&ty,
2161                           pvx,pvy,pkx,pky,pwx,pwy,
2162                           vklmn,kklmn,wklmn);
2163
2164    }
2165    double result=0.;
```

```
2166
2167    double s1=kklmn[0]/2.;
2168    double s2=(kklmn[0]+kklmn[1331])/4.;
2169    double s3=(kklmn[0]+kklmn[121])/4.;
2170    double s4=(kklmn[0]+kklmn[121]+kklmn[1331]+kklmn[1452])/8.;
2171    if(psi_y1pos==0 && psi_y2pos==0) //psiA
2172      result=s1-s2-s3+s4;
2173
2174    if(psi_y1pos==1 && psi_y2pos==0)//psiB
2175      result=s2-s4;
2176
2177    if(psi_y1pos==1 && psi_y2pos==1)//psiC
2178      result=s4;
2179
2180    if(psi_y1pos==0 && psi_y2pos==1)//psiD
2181      result=s3-s4;
2182
2183    free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
2184    free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
2185    free(vklmn);free(kklmn);free(wklmn);
2186
2187    return result;
2188  }
2189
2190  double compute_entry_DLP_linear(Triangulation3D& row_mesh,Triangulation3D& col_mesh,
2191                                  int i,int j)
2192  {
2193    Matrix x_rot=Matrix(3);
2194    x_rot.set(1,1,1);
2195    x_rot.set(1,2,0);
2196    x_rot.set(1,3,0);
2197    x_rot.set(2,1,0);
2198    x_rot.set(2,2,0);
2199    x_rot.set(2,3,-1);
2200    x_rot.set(3,1,0);
2201    x_rot.set(3,2,1);
2202    x_rot.set(3,3,0);
2203    Matrix y_rot=Matrix(3);
2204    y_rot.set(1,1,0);
2205    y_rot.set(1,2,0);
2206    y_rot.set(1,3,1);
2207    y_rot.set(2,1,0);
2208    y_rot.set(2,2,1);
2209    y_rot.set(2,3,0);
2210    y_rot.set(3,1,-1);
2211    y_rot.set(3,2,0);
2212    y_rot.set(3,3,0);
2213    Matrix z_rot=Matrix(3);
2214    z_rot.set(1,1,0);
2215    z_rot.set(1,2,-1);
2216    z_rot.set(1,3,0);
2217    z_rot.set(2,1,1);
2218    z_rot.set(2,2,0);
2219    z_rot.set(2,3,0);
2220    z_rot.set(3,1,0);
2221    z_rot.set(3,2,0);
2222    z_rot.set(3,3,1);
2223
2224    double* bx=(double*)malloc(3*sizeof(double));
2225    double* dx1=(double*)malloc(3*sizeof(double));
2226    double* dx2=(double*)malloc(3*sizeof(double));
2227    double* nx=(double*)malloc(3*sizeof(double));
2228    int tx=4;
```

```
2229
2230    double* by=(double*)malloc(3*sizeof(double));
2231    double* dy1=(double*)malloc(3*sizeof(double));
2232    double* dy2=(double*)malloc(3*sizeof(double));
2233    double* ny=(double*)malloc(3*sizeof(double));
2234    int ty=4;
2235
2236    int* pvx=(int*)malloc(2*sizeof(int));
2237    pvx[0]=-1;pvx[1]=-1;
2238    int* pvy=(int*)malloc(2*sizeof(int));
2239    pvy[0]=-1;pvy[1]=-1;
2240    int* pkx=(int*)malloc(2*sizeof(int));
2241    pkx[0]=1;pkx[1]=1;
2242    int* pky=(int*)malloc(2*sizeof(int));
2243    pky[0]=1;pky[1]=1;
2244    int* pwx=(int*)malloc(2*sizeof(int));
2245    pwx[0]=-1;pwx[1]=-1;
2246    int* pwy=(int*)malloc(2*sizeof(int));
2247    pwy[0]=-1;pwy[1]=-1;
2248
2249    double* vklmn=(double*)malloc(sizeof(double));
2250    double* kklmn=(double*)malloc(1465*sizeof(double));
2251    double* wklmn=(double*)malloc(sizeof(double));
2252    for(int k=0;k<1465;++k)
2253      kklmn[k]=0.;
2254
2255    int I=i/4;
2256    int J=j/4;
2257
2258    Shape* R1=row_mesh[I];
2259    Shape* R2=col_mesh[J];
2260
2261    if(I==J)
2262      return R1->gram_det()/32.;
2263
2264    I=i;
2265    J=j;
2266    int psi1_xpos=0;
2267    int psi1_ypos=0;
2268    int psi2_xpos=0;
2269    int psi2_ypos=0;
2270    std::vector<Point3D> v1=R1->get_vertices();
2271    std::vector<Point3D> v2=R2->get_vertices();
2272    bool is_parallel=false;
2273    switch (check_parallel(*R1,*R2))
2274    {
2275      case XYXY:
2276        is_parallel=true;
2277        break;
2278      case XYYZ:
2279        for(int i=0;i<4;++i)
2280        {
2281          v1[i]=MVM(z_rot,v1[i]);
2282          v2[i]=MVM(z_rot,v2[i]);
2283        }
2284        break;
2285      case XYZX:
2286        break;
2287      case YZXY:
2288        for(int i=0;i<4;++i)
2289        {
2290          v1[i]=MVM(y_rot,v1[i]);
2291          v2[i]=MVM(y_rot,v2[i]);
```

```
2292        }
2293        for(int i=0;i<4;++i)
2294        {
2295          v1[i]=MVM(z_rot,v1[i]);
2296          v2[i]=MVM(z_rot,v2[i]);
2297        }
2298        break;
2299      case YZYZ:
2300        is_parallel=true;
2301        for(int i=0;i<4;++i)
2302        {
2303          v1[i]=MVM(y_rot,v1[i]);
2304          v2[i]=MVM(y_rot,v2[i]);
2305        }
2306        break;
2307      case YZZX:
2308        for(int i=0;i<4;++i)
2309        {
2310          v1[i]=MVM(y_rot,v1[i]);
2311          v2[i]=MVM(y_rot,v2[i]);
2312        }
2313        break;
2314      case ZXXY:
2315        for(int i=0;i<4;++i)
2316        {
2317          v1[i]=MVM(x_rot,v1[i]);
2318          v2[i]=MVM(x_rot,v2[i]);
2319        }
2320        break;
2321      case ZXYZ:
2322        for(int i=0;i<4;++i)
2323        {
2324          v1[i]=MVM(x_rot,v1[i]);
2325          v2[i]=MVM(x_rot,v2[i]);
2326        }
2327        for(int i=0;i<4;++i)
2328        {
2329          v1[i]=MVM(z_rot,v1[i]);
2330          v2[i]=MVM(z_rot,v2[i]);
2331        }
2332        break;
2333      case ZXZX:
2334        is_parallel=true;
2335        for(int i=0;i<4;++i)
2336        {
2337          v1[i]=MVM(x_rot,v1[i]);
2338          v2[i]=MVM(x_rot,v2[i]);
2339        }
2340        break;
2341      default:
2342        std::cout<<"Wrong␣Case!\n";
2343        exit(1);
2344        break;
2345    }
2346    double x1upper=0.;
2347    double x1lower=0.;
2348    double x2upper=0.;
2349    double x2lower=0.;
2350    double y1upper=0.;
2351    double y1lower=0.;
2352    double y2upper=0.;
2353    double y2lower=0.;
2354    if(is_parallel)
```

```
2355    {
2356      x1upper=v1[0].x();
2357      x1lower=v1[0].x();
2358      x2upper=v1[0].y();
2359      x2lower=v1[0].y();
2360      y1upper=v2[0].x();
2361      y1lower=v2[0].x();
2362      y2upper=v2[0].y();
2363      y2lower=v2[0].y();
2364        //check which basis functions are meant. xpos bzw. ypos denote
2365        //weather the basis-function has its peak left (xpos=0) or right
2366        //(xpos=1) and front (ypos=0) or back (ypos=1)
2367      for(int i=1;i<4;++i)
2368      {
2369        if(v1[i].x()>x1upper)
2370          x1upper=v1[i].x();
2371        if(v1[i].x()<x1lower)
2372          x1lower=v1[i].x();
2373        if(v1[i].y()>x2upper)
2374          x2upper=v1[i].y();
2375        if(v1[i].y()<x2lower)
2376          x2lower=v1[i].y();
2377        if(v2[i].x()>y1upper)
2378          y1upper=v2[i].x();
2379        if(v2[i].x()<y1lower)
2380          y1lower=v2[i].x();
2381        if(v2[i].y()>y2upper)
2382          y2upper=v2[i].y();
2383        if(v2[i].y()<y2lower)
2384          y2lower=v2[i].y();
2385      }
2386      switch (I%4)
2387      {
2388        case 0:
2389          if(v1[0].x()==x1lower)
2390            psi1_xpos=0;
2391          else
2392            psi1_xpos=1;
2393          if(v1[0].y()==x2lower)
2394            psi1_ypos=0;
2395          else
2396            psi1_ypos=1;
2397          break;
2398        case 1:
2399          if(v1[1].x()==x1lower)
2400            psi1_xpos=0;
2401          else
2402            psi1_xpos=1;
2403          if(v1[1].y()==x2lower)
2404            psi1_ypos=0;
2405          else
2406            psi1_ypos=1;
2407          break;
2408        case 2:
2409          if(v1[2].x()==x1lower)
2410            psi1_xpos=0;
2411          else
2412            psi1_xpos=1;
2413          if(v1[2].y()==x2lower)
2414            psi1_ypos=0;
2415          else
2416            psi1_ypos=1;
2417          break;
```

```
2418        case 3:
2419          if(v1[3].x()==x1lower)
2420            psi1_xpos=0;
2421          else
2422            psi1_xpos=1;
2423          if(v1[3].y()==x2lower)
2424            psi1_ypos=0;
2425          else
2426            psi1_ypos=1;
2427          break;
2428      }
2429      switch(J%4)
2430      {
2431        case 0:
2432          if(v2[0].x()==y1lower)
2433            psi2_xpos=0;
2434          else
2435            psi2_xpos=1;
2436          if(v2[0].y()==y2lower)
2437            psi2_ypos=0;
2438          else
2439            psi2_ypos=1;
2440          break;
2441        case 1:
2442          if(v2[1].x()==y1lower)
2443            psi2_xpos=0;
2444          else
2445            psi2_xpos=1;
2446          if(v2[1].y()==y2lower)
2447            psi2_ypos=0;
2448          else
2449            psi2_ypos=1;
2450          break;
2451        case 2:
2452          if(v2[2].x()==y1lower)
2453            psi2_xpos=0;
2454          else
2455            psi2_xpos=1;
2456          if(v2[2].y()==y2lower)
2457            psi2_ypos=0;
2458          else
2459            psi2_ypos=1;
2460          break;
2461        case 3:
2462          if(v2[3].x()==y1lower)
2463            psi2_xpos=0;
2464          else
2465            psi2_xpos=1;
2466          if(v2[3].y()==y2lower)
2467            psi2_ypos=0;
2468          else
2469            psi2_ypos=1;
2470          break;
2471      }
2472      bx[0]=x1lower;
2473      bx[1]=x2lower;
2474      bx[2]=v1[0].z();
2475
2476      dx1[0]=x1upper-x1lower;
2477      dx1[1]=0.;
2478      dx1[2]=0.;
2479      dx2[0]=0.;
2480      dx2[1]=x2upper-x2lower;
```

```
2481      dx2[2]=0.;
2482
2483      by[0]=y1lower;
2484      by[1]=y2lower;
2485      by[2]=v2[0].z();
2486
2487      dy1[0]=y1upper-y1lower;
2488      dy1[1]=0.;
2489      dy1[2]=0.;
2490      dy2[0]=0.;
2491      dy2[1]=y2upper-y2lower;
2492      dy2[2]=0.;
2493
2494      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
2495      normalize(n);
2496      nx[0]=n.x();
2497      nx[1]=n.y();
2498      nx[2]=n.z();
2499
2500      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
2501      normalize(n);
2502      ny[0]=n.x();
2503      ny[1]=n.y();
2504      ny[2]=n.z();
2505
2506      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
2507                           by,dy1,dy2,ny,&ty,
2508                           pvx,pvy,pkx,pky,pwx,pwy,
2509                           vklmn,kklmn,wklmn);
2510    }
2511    else
2512    {
2513      x1upper=v1[0].x();
2514      x1lower=v1[0].x();
2515      x2upper=v1[0].y();
2516      x2lower=v1[0].y();
2517      y1upper=v2[0].x();
2518      y1lower=v2[0].x();
2519      y2upper=v2[0].z();
2520      y2lower=v2[0].z();
2521      for(int i=1;i<4;++i)
2522      {
2523        if(v1[i].x()>x1upper)
2524          x1upper=v1[i].x();
2525        if(v1[i].x()<x1lower)
2526          x1lower=v1[i].x();
2527        if(v1[i].y()>x2upper)
2528          x2upper=v1[i].y();
2529        if(v1[i].y()<x2lower)
2530          x2lower=v1[i].y();
2531        if(v2[i].x()>y1upper)
2532          y1upper=v2[i].x();
2533        if(v2[i].x()<y1lower)
2534          y1lower=v2[i].x();
2535        if(v2[i].z()>y2upper)
2536          y2upper=v2[i].z();
2537        if(v2[i].z()<y2lower)
2538          y2lower=v2[i].z();
2539      }
2540      switch (I%4)
2541      {
2542        case 0:
2543          if(v1[0].x()==x1lower)
```

```
2544            psi1_xpos=0;
2545          else
2546            psi1_xpos=1;
2547          if(v1[0].y()==x2lower)
2548            psi1_ypos=0;
2549          else
2550            psi1_ypos=1;
2551          break;
2552        case 1:
2553          if(v1[1].x()==x1lower)
2554            psi1_xpos=0;
2555          else
2556            psi1_xpos=1;
2557          if(v1[1].y()==x2lower)
2558            psi1_ypos=0;
2559          else
2560            psi1_ypos=1;
2561          break;
2562        case 2:
2563          if(v1[2].x()==x1lower)
2564            psi1_xpos=0;
2565          else
2566            psi1_xpos=1;
2567          if(v1[2].y()==x2lower)
2568            psi1_ypos=0;
2569          else
2570            psi1_ypos=1;
2571          break;
2572        case 3:
2573          if(v1[3].x()==x1lower)
2574            psi1_xpos=0;
2575          else
2576            psi1_xpos=1;
2577          if(v1[3].y()==x2lower)
2578            psi1_ypos=0;
2579          else
2580            psi1_ypos=1;
2581          break;
2582      }
2583      switch(J%4)
2584      {
2585        case 0:
2586          if(v2[0].x()==y1lower)
2587            psi2_xpos=0;
2588          else
2589            psi2_xpos=1;
2590          if(v2[0].z()==y2lower)
2591            psi2_ypos=0;
2592          else
2593            psi2_ypos=1;
2594          break;
2595        case 1:
2596          if(v2[1].x()==y1lower)
2597            psi2_xpos=0;
2598          else
2599            psi2_xpos=1;
2600          if(v2[1].z()==y2lower)
2601            psi2_ypos=0;
2602          else
2603            psi2_ypos=1;
2604          break;
2605        case 2:
2606          if(v2[2].x()==y1lower)
```

```
2607            psi2_xpos=0;
2608          else
2609            psi2_xpos=1;
2610          if(v2[2].z()==y2lower)
2611            psi2_ypos=0;
2612          else
2613            psi2_ypos=1;
2614          break;
2615        case 3:
2616          if(v2[3].x()==y1lower)
2617            psi2_xpos=0;
2618          else
2619            psi2_xpos=1;
2620          if(v2[3].z()==y2lower)
2621            psi2_ypos=0;
2622          else
2623            psi2_ypos=1;
2624          break;
2625      }
2626      bx[0]=x1lower;
2627      bx[1]=x2lower;
2628      bx[2]=v1[0].z();
2629
2630      dx1[0]=x1upper-x1lower;
2631      dx1[1]=0.;
2632      dx1[2]=0.;
2633      dx2[0]=0.;
2634      dx2[1]=x2upper-x2lower;
2635      dx2[2]=0.;
2636
2637      by[0]=y1lower;
2638      by[1]=v2[0].y();
2639      by[2]=y2lower;
2640
2641      dy1[0]=y1upper-y1lower;
2642      dy1[1]=0.;
2643      dy1[2]=0.;
2644      dy2[0]=0.;
2645      dy2[1]=0.;
2646      dy2[2]=y2upper-y2lower;
2647
2648      Vector n=vectorproduct(v1[1]-v1[0],v1[3]-v1[0]);
2649      normalize(n);
2650      nx[0]=n.x();
2651      nx[1]=n.y();
2652      nx[2]=n.z();
2653
2654      n=vectorproduct(v2[1]-v2[0],v2[3]-v2[0]);
2655      normalize(n);
2656      ny[0]=0.;
2657      ny[1]=-n.y();
2658      ny[2]=0.;
2659
2660      liblap3_MP_lapinteg3(bx,dx1,dx2,nx,&tx,
2661                           by,dy1,dy2,ny,&ty,
2662                           pvx,pvy,pkx,pky,pwx,pwy,
2663                           vklmn,kklmn,wklmn);
2664    }
2665
2666    double s1=kklmn[0]/2.;
2667    double s2=(kklmn[0]+kklmn[121])/4.;
2668    double s3=(kklmn[0]+kklmn[1331])/4.;
2669    double s4=(kklmn[0]+kklmn[121]+kklmn[1331]+kklmn[1452])/8.;
```

```
2670    double s5=(kklmn[0]+kklmn[1])/4.;
2671    double s6=(kklmn[0]+kklmn[1]+kklmn[121]+kklmn[122])/8.;
2672    double s7=(kklmn[0]+kklmn[1]+kklmn[1331]+kklmn[1332])/8.;
2673    double s8=(kklmn[0]+kklmn[1]+kklmn[121]+kklmn[122]+kklmn[1331]+kklmn[1332]+
2674                kklmn[1452]+kklmn[1453])/16.;
2675    double s9=(kklmn[0]+kklmn[11])/4.;
2676    double s10=(kklmn[0]+kklmn[11]+kklmn[121]+kklmn[132])/8.;
2677    double s11=(kklmn[0]+kklmn[11]+kklmn[1331]+kklmn[1342])/8.;
2678    double s12=(kklmn[0]+kklmn[11]+kklmn[121]+kklmn[132]+kklmn[1331]+kklmn[1342]+
2679                kklmn[1452]+kklmn[1463])/16.;
2680    double s13=(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12])/8.;
2681    double s14=(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12]+kklmn[121]+kklmn[122]+kklmn
        [132]+
2682                kklmn[133])/16.;
2683    double s15=(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12]+kklmn[1331]+kklmn[1332]+
2684                kklmn[1342]+kklmn[1343])/16.;
2685    double s16=(kklmn[0]+kklmn[1]+kklmn[11]+kklmn[12]+kklmn[121]+kklmn[122]+
2686                kklmn[132]+kklmn[133]+
2687                kklmn[1331]+kklmn[1332]+kklmn[1342]+kklmn[1343]+kklmn[1452]+
2688                kklmn[1453]+kklmn[1463]+kklmn[1464])/32.;
2689
2690    double result=0.;
2691
2692    if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==0) //psiA and psiA'
2693      result=s1-s2-s3+s4-s5+s6+s7-s8-s9+s10+s11-s12+s13-s14-s15+s16;
2694
2695    if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==0)//psiA and psiB'
2696      result=s2-s4-s6+s8-s10+s12+s14-s16;
2697
2698    if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==1)//psiA and psiC'
2699      result=s4-s8-s12+s16;
2700
2701    if(psi1_xpos==0 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==1)//psiA and psiD'
2702      result=s3-s7-s11+s15-s4+s8+s12-s16;
2703
2704    if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==0)//psiB and psiA'
2705      result=s5-s6-s7+s8-s13+s14+s15-s16;
2706
2707    if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==0)//psiB and psiB'
2708      result=s6-s8-s14+s16;
2709
2710    if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==1 && psi2_ypos==1)//psiB and psiC'
2711      result=s8-s16;
2712
2713    if(psi1_xpos==1 && psi1_ypos==0 && psi2_xpos==0 && psi2_ypos==1)//psiB and psiD'
2714      result=s7-s8-s15+s16;
2715
2716    if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==0)//psiC and psiA'
2717      result=s13-s14-s15+s16;
2718
2719    if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==0)//psiC and psiB'
2720      result=s14-s16;
2721
2722    if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==1)//psiC and psiC'
2723      result=s16;
2724
2725    if(psi1_xpos==1 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==1)//psiC and psiD'
2726      result=s15-s16;
2727
2728    if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==0)//psiD and psiA'
2729      result=s9-s10-s11+s12-s13+s14+s15-s16;
2730
2731    if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==0)//psiD and pisB'
```

```
2732      result=s10-s12-s14+s16;
2733
2734    if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==1 && psi2_ypos==1)//psiD and psiC'
2735      result=s12-s16;
2736
2737    if(psi1_xpos==0 && psi1_ypos==1 && psi2_xpos==0 && psi2_ypos==1)//psiD and psiD'
2738      result=s11-s12-s15+s16;
2739
2740    free(bx);free(dx1);free(dx2);free(nx);free(by);free(dy1);free(dy2);free(ny);
2741    free(pvx);free(pvy);free(pkx);free(pky);free(pwx);free(pwy);
2742    free(vklmn);free(kklmn);free(wklmn);
2743
2744    return result;
2745  }
2746
2747  //*************************************************************************//
2748  // The ACA-Algorithm for general and symmetrical matrices                  //
2749  //*************************************************************************//
2750  //To understand these functions you need a deep notion of the structure of
2751  //a HH-Matrix. The way ACA works is explained in the lecture notes from the
2752  //Max-Planck-Institute. However this function aonly calls the subroutine from
2753  //the HLib, where an approximative filling should be done.
2754  void fill_supermatrix_aca(supermatrix* A,const cluster* row,const cluster* col,
2755                            meshes_for_aca* meshdat,
2756                            double (*compute_entry)(int i,int j,void* data),
2757                            double aca_eps)
2758  {
2759    clustertree* rct=(clustertree*)meshdat->row_mesh->data;
2760    clustertree* cct=(clustertree*)meshdat->col_mesh->data;
2761    fullmatrix* f=A->f;
2762    rkmatrix* r=A->r;
2763    int rank;
2764
2765    if(f!=NULL)
2766    {
2767      for(int i=0;i<row->size;++i)
2768        for(int j=0;j<col->size;++j)
2769          f->e[i+j*row->size]=
2770              compute_entry(row->start+i,col->start+j,(void*) meshdat);
2771      return;
2772    }
2773
2774    if(r!=NULL)
2775    {
2776      if(r->rows>r->cols)
2777        rank=r->cols;
2778      else
2779        rank=r->rows;
2780
2781      if(rank>ACA_KMAX)
2782        rank=(int)ACA_KMAX;
2783
2784  //    rank+=10;
2785      r->k=rank;
2786      r->a=allocate_matrix(r->rows,rank);
2787      r->b=allocate_matrix(r->cols,rank);
2788
2789      r->kt=newaca_fill_block(r->a,r->b,r->rows,r->cols,row->start,col->start,
2790                              compute_entry,(void*)meshdat,rank,aca_eps,
2791                              ACA_STRATEGY);
2792
2793      r->a=(double*)realloc(r->a,r->rows*r->kt*sizeof(double));
2794      r->b=(double*)realloc(r->b,r->cols*r->kt*sizeof(double));
```

```
2795      r->k=r->kt;
2796
2797      return;
2798    }
2799
2800    if(row->sons>0)
2801      if(col->sons>0)
2802        for(int i=0;i<row->sons;++i)
2803          for(int j=0;j<col->sons;++j)
2804    {
2805      int s=i+j*row->sons;
2806      fill_supermatrix_aca(A->s[s],row->son[i],
2807                           col->son[j],meshdat,compute_entry,aca_eps);
2808    }
2809    else
2810      for(int i=0;i<row->sons;++i)
2811        fill_supermatrix_aca(A->s[i],row->son[i],col,meshdat,compute_entry,aca_eps);
2812    else
2813      for(int i=0;i<col->sons;++i)
2814        fill_supermatrix_aca(A->s[i],row,col->son[i],meshdat,compute_entry,aca_eps);
2815  }
2816
2817
2818  void fill_symetric_supermatrix_aca(supermatrix* A,supermatrix* AT,
2819                                     const cluster* row,const cluster* col,
2820                                     meshes_for_aca* meshdat,
2821                                     double (*compute_entry)(int i,int j,void* data),
2822                                     double epsilon)
2823  {
2824    clustertree* rct=(clustertree*)meshdat->row_mesh->data;
2825    clustertree* cct=(clustertree*)meshdat->col_mesh->data;
2826    fullmatrix* f=A->f;
2827    rkmatrix* r=A->r;
2828    int rank;
2829
2830    if(f!=NULL)
2831    {
2832      for(int i=0;i<row->size;++i)
2833        for(int j=0;j<col->size;++j)
2834          f->e[i+j*row->size]=
2835              compute_entry(row->start+i,col->start+j,(void*) meshdat);
2836      if(row!=col)
2837      {
2838        for(int i=0;i<col->size;++i)
2839          for(int j=0;j<row->size;++j)
2840            AT->f->e[i+j*col->size]=f->e[j+i*row->size];
2841      }
2842      return;
2843    }
2844
2845    if(r!=NULL)
2846    {
2847      if(r->rows>r->cols)
2848        rank=r->cols;
2849      else
2850        rank=r->rows;
2851
2852      if(rank>ACA_KMAX)
2853        rank=(int)ACA_KMAX;
2854
2855      r->k=rank;
2856      r->a=allocate_matrix(r->rows,rank);
2857      r->b=allocate_matrix(r->cols,rank);
```

```
2858
2859      r->kt=newaca_fill_block(r->a,r->b,r->rows,r->cols,row->start,col->start,
2860                              compute_entry,(void*)meshdat,rank,epsilon,
2861                              ACA_STRATEGY);
2862
2863      r->a=(double*)realloc(r->a,r->rows*r->kt*sizeof(double));
2864      r->b=(double*)realloc(r->b,r->cols*r->kt*sizeof(double));
2865      r->k=r->kt;
2866
2867      if(row!=col)
2868      {
2869        if(r->rows>r->cols)
2870          rank=r->cols;
2871        else
2872          rank=r->rows;
2873
2874        if(rank>ACA_KMAX)
2875          rank=(int)ACA_KMAX;
2876
2877        AT->r->k=rank;
2878        AT->r->a=allocate_matrix(r->cols,rank);
2879        AT->r->b=allocate_matrix(r->rows,rank);
2880        AT->r->kt=r->k;
2881
2882        AT->r->a=(double*)realloc(AT->r->a,AT->r->rows*AT->r->kt*sizeof(double));
2883        AT->r->b=(double*)realloc(AT->r->b,AT->r->cols*AT->r->kt*sizeof(double));
2884        AT->r->k=AT->r->kt;
2885        for(int i=0;i<AT->r->rows*AT->r->k;++i)
2886          AT->r->a[i]=r->b[i];
2887        for(int i=0;i<AT->r->cols*AT->r->k;++i)
2888          AT->r->b[i]=r->a[i];
2889        //       compress_supermatrix(AT,epsilon,50);
2890      }
2891      //    compress_supermatrix(A,epsilon,50);
2892      return;
2893    }
2894
2895    if(row->sons>0 && col->sons>0)
2896    {
2897      fill_symetric_supermatrix_aca(A->s[0],AT->s[0],row->son[0],
2898                                    col->son[0],meshdat,compute_entry,epsilon);
2899      fill_symetric_supermatrix_aca(A->s[3],AT->s[3],row->son[1],
2900                                    col->son[1],meshdat,compute_entry,epsilon);
2901      fill_symetric_supermatrix_aca(A->s[2],AT->s[1],row->son[0],
2902                                    col->son[1],meshdat,compute_entry,epsilon);
2903      if(row!=col)
2904        fill_symetric_supermatrix_aca(A->s[1],AT->s[2],row->son[1],
2905                                      col->son[0],meshdat,compute_entry,epsilon);
2906    }
2907    else if(row->sons>0)
2908      for(int i=0;i<row->sons;++i)
2909        fill_symetric_supermatrix_aca(A->s[i],AT->s[i],row->son[i],
2910                                      col,meshdat,compute_entry,epsilon);
2911    else
2912      for(int i=0;i<col->sons;++i)
2913        fill_symetric_supermatrix_aca(A->s[i],AT->s[i],row,
2914                                      col->son[i],meshdat,compute_entry,epsilon);
2915  }
2916
2917  //*************************************************************************//
2918  // Functions of SLP or directly concerning SLP                             //
2919  //*************************************************************************//
2920
```

```
2921 //dia_prec builds a HH-Matrix with identical structure, that can be used by
2922 //gmres as preconditioner. It is diagonal preconditioning
2923 void diag_prec(supermatrix* a,supermatrix* b)
2924 {
2925   int i,j,k,nu;
2926   double* m;
2927   int rows,cols;
2928   double div;
2929
2930   assert(a!=0x0);
2931   if(a->s!=0x0)
2932   {
2933     diag_prec(a->s[0],b->s[0]);
2934     diag_prec(a->s[3],b->s[3]);
2935   }
2936   else{
2937     assert(a->f!=0x0);
2938     assert(a->rows==a->cols);
2939     rows = a->rows;
2940     cols = a->cols;
2941
2942     for(k=0; k<rows && k<rows; ++k)
2943       if(b->f->e[k+k*rows]!=0)
2944          a->f->e[k+k*rows]=1./b->f->e[k+k*rows];
2945       else
2946       {
2947         std::cout<<"test\n";
2948         a->f->e[k+k*rows]=1.;
2949       }
2950   }
2951 }
2952
2953 SLP::~SLP()
2954 {
2955   del_supermatrix(A);
2956   if(rct!=cct)
2957     del_clustertree(rct);
2958   del_clustertree(cct);
2959 }
2960
2961 void SLP::build(Triangulation3D& row_mesh,Triangulation3D& col_mesh,
2962                 function_type ansatz,double epsilon)
2963 {
2964   if(ansatz==CONSTANT)
2965   {
2966     ansatz_functions=CONSTANT;
2967     clusterfactory* cf=NULL;
2968     int cleaf=30;
2969     cf=CreateClusterFactory(row_mesh);
2970     int N=row_mesh.size();
2971     rct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
2972     del_clusterfactory(cf);
2973     cf=CreateClusterFactory(col_mesh);
2974     N=col_mesh.size();
2975     cct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
2976     del_clusterfactory(cf);
2977     row_mesh.data=(void*)rct;
2978     col_mesh.data=(void*)cct;
2979     meshes_for_aca* meshdat=(meshes_for_aca*)malloc(sizeof(meshes_for_aca));
2980     meshdat->row_mesh=&row_mesh;
2981     meshdat->col_mesh=&col_mesh;
2982     if(epsilon<=0)
2983       A=build_full_supermatrix(rct->root,cct->root);
```

```
2984      else
2985        A=build_supermatrix_from_cluster(rct->root,cct->root,1,cleaf,HLIB_MINADM);
2986      fill_supermatrix_aca(A,rct->root,cct->root,meshdat,compute_entry_SLP_const,
2987                           epsilon);
2988    }
2989    else if(ansatz==LINEAR)
2990    {
2991      ansatz_functions=LINEAR;
2992      clusterfactory* cf=NULL;
2993      int cleaf=30;
2994      cf=CreateClusterFactoryLinear(row_mesh);
2995      int N=4*row_mesh.size();
2996      rct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
2997      del_clusterfactory(cf);
2998      cf=CreateClusterFactoryLinear(col_mesh);
2999      N=4*col_mesh.size();
3000      cct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
3001      del_clusterfactory(cf);
3002      row_mesh.data=(void*)rct;
3003      col_mesh.data=(void*)cct;
3004      meshes_for_aca* meshdat=(meshes_for_aca*)malloc(sizeof(meshes_for_aca));
3005      meshdat->row_mesh=&row_mesh;
3006      meshdat->col_mesh=&col_mesh;
3007      if(epsilon<=0)
3008        A=build_full_supermatrix(rct->root,cct->root);
3009      else
3010        A=build_supermatrix_from_cluster(rct->root,cct->root,1,cleaf,HLIB_MINADM);
3011      fill_supermatrix_aca(A,rct->root,cct->root,meshdat,compute_entry_SLP_linear,
3012                           epsilon);
3013    }
3014    else if(ansatz==LINCONST)
3015    {
3016      ansatz_functions=LINCONST;
3017      clusterfactory* cf=NULL;
3018      int cleaf=30;
3019      cf=CreateClusterFactoryLinear(row_mesh);
3020      int N=4*row_mesh.size();
3021      rct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
3022      del_clusterfactory(cf);
3023      cf=CreateClusterFactory(col_mesh);
3024      N=col_mesh.size();
3025      cct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
3026      del_clusterfactory(cf);
3027      row_mesh.data=(void*)rct;
3028      col_mesh.data=(void*)cct;
3029      meshes_for_aca* meshdat=(meshes_for_aca*)malloc(sizeof(meshes_for_aca));
3030      meshdat->row_mesh=&row_mesh;
3031      meshdat->col_mesh=&col_mesh;
3032      if(epsilon<=0)
3033        A=build_full_supermatrix(rct->root,cct->root);
3034      else
3035        A=build_supermatrix_from_cluster(rct->root,cct->root,1,cleaf,HLIB_MINADM);
3036      fill_supermatrix_aca(A,rct->root,cct->root,meshdat,compute_SLP_linconst,
3037                           epsilon);
3038    }
3039  }
3040
3041  void SLP::build(Triangulation3D& mesh,function_type ansatz,double epsilon)
3042  {
3043    if(ansatz==CONSTANT)
3044    {
3045      ansatz_functions=CONSTANT;
3046      clusterfactory* cf=NULL;
```

```
3047      int cleaf=30;
3048      cf=CreateClusterFactory(mesh);
3049      int N=mesh.size();
3050      rct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
3051      del_clusterfactory(cf);
3052      cct=rct;
3053      mesh.data=(void*)rct;
3054      meshes_for_aca* meshdat=(meshes_for_aca*)malloc(sizeof(meshes_for_aca));
3055      meshdat->row_mesh=&mesh;
3056      meshdat->col_mesh=&mesh;
3057      if(epsilon<=0)
3058        A=build_full_supermatrix(rct->root,cct->root);
3059      else
3060        A=build_supermatrix_from_cluster(rct->root,cct->root,1,cleaf,HLIB_MINADM);
3061      fill_symetric_supermatrix_aca(A,A,rct->root,rct->root,meshdat,
3062                                    compute_entry_SLP_const,epsilon);
3063    }
3064    else if(ansatz==LINEAR)
3065    {
3066      ansatz_functions=LINEAR;
3067      clusterfactory* cf=NULL;
3068      int cleaf=30;
3069      cf=CreateClusterFactoryLinear(mesh);
3070      int N=4*mesh.size();
3071      rct=create_clustertree(cf,HLIB_GEOMETRIC,cleaf,N*N);
3072      del_clusterfactory(cf);
3073      cct=rct;
3074      mesh.data=(void*)rct;
3075      meshes_for_aca* meshdat=(meshes_for_aca*)malloc(sizeof(meshes_for_aca));
3076      meshdat->row_mesh=&mesh;
3077      meshdat->col_mesh=&mesh;
3078      if(epsilon<=0)
3079        A=build_full_supermatrix(rct->root,cct->root);
3080      else
3081        A=build_supermatrix_from_cluster(rct->root,cct->root,1,cleaf,HLIB_MINADM);
3082      fill_symetric_supermatrix_aca(A,A,rct->root,cct->root,meshdat,
3083                                    compute_entry_SLP_linear,epsilon);
3084    }
3085    else
3086    {
3087      std::cout<<"FATAL␣ERROR:␣ANSATZ-TYPE␣NOT␣KNOWN␣IN␣FUNCTION␣BUILD\n";
3088      exit(1);
3089    }
3090  }
3091
3092  supermatrix* SLP::build_preconditioner()
3093  {
3094      supermatrix* M = copystructure_supermatrix(A);
3095      diag_prec(M,A);
3096      return M;
3097  }
3098
3099  double* SLP::reorder(double* v,clustertree* ct)
3100  {
3101    double* temp=(double*)malloc(ct->root->size*sizeof(double));
3102    for(int i=0;i<ct->root->size;++i)
3103      temp[i]=v[ct->idx2dof[i]];
3104
3105    free(v);
3106    v=temp;
3107    temp=NULL;
3108    return v;
3109  }
```

```
3110
3111 double* SLP::disorder(double* v,clustertree* ct)
3112 {
3113   double* temp=(double*)malloc(ct->root->size*sizeof(double));
3114   for(int i=0;i<ct->root->size;++i)
3115     temp[i]=v[ct->dof2idx[i]];
3116
3117   free(v);
3118   v=temp;
3119   temp=NULL;
3120   return v;
3121 }
3122
3123 double* SLP::operator* (double* x)
3124 {
3125   double* x_temp=(double*)malloc(cct->ndof*sizeof(double));
3126   for(int i=0;i<cct->ndof;++i)
3127     x_temp[i]=x[i];
3128   x_temp=disorder(x_temp,cct);
3129   double* b=(double*)malloc(rct->ndof*sizeof(double));
3130   for(int i=0;i<rct->ndof;++i)
3131     b[i]=0.;
3132   eval_supermatrix(A,x_temp,b);
3133   b=reorder(b,rct);
3134   free(x_temp);
3135   return b;
3136 }
3137
3138 double* SLP::operator/ (double* b)
3139 {
3140   double* b_temp=(double*)malloc(rct->ndof*sizeof(double));
3141   for(int i=0;i<rct->ndof;++i)
3142     b_temp[i]=b[i];
3143   b_temp=disorder(b_temp,rct);
3144   double* x=(double*)malloc(cct->ndof*sizeof(double));
3145   for(int i=0;i<cct->ndof;++i)
3146     x[i]=0.;
3147   if(A->f!=NULL)
3148   {
3149     Matrix new_mat=Matrix(A->f->rows,A->f->cols);
3150     for(int i=0;i<A->f->rows*A->f->cols;++i)
3151       new_mat.data[i]=A->f->e[i];
3152     free(x);
3153     x=solveGauss(new_mat,b_temp);
3154     x=reorder(x,cct);
3155     free(b_temp);
3156     return x;
3157   }
3158   supermatrix* M_precond = build_preconditioner();
3159
3160   solve_gmres_supermatrix(A, b_temp, x, 1e-14, 1000, M_precond,
3161                           HLIB_PREC_INVERSE,NULL,HLIB_EVAL_DEFAULT,1);
3162   del_supermatrix(M_precond);
3163   x=reorder(x,cct);
3164   free(b_temp);
3165   return x;
3166 }
3167
3168 //*************************************************************************//
3169 // Implementation of functions in DLP                                      //
3170 //*************************************************************************//
3171 DLP_RHS::~DLP_RHS()
3172 {
```

```
3173
3174 }
3175
3176 void DLP_RHS::build(Triangulation3D& row_mesh,Triangulation3D& col_mesh,
3177                     function_type ansatz)
3178 {
3179   if(ansatz==CONSTANT)
3180   {
3181     double* new_data=(double*)malloc(row_mesh.size()*col_mesh.size()*
3182                                      sizeof(double));
3183     A=Matrix(row_mesh.size(),col_mesh.size());
3184     for(int i=0;i<row_mesh.size();++i)
3185       for(int j=0;j<col_mesh.size();++j)
3186       {
3187         double value=compute_entry_DLP_const(row_mesh,col_mesh,i,j);
3188         new_data[j*row_mesh.size()+i]=value;
3189       }
3190     A.data=new_data;
3191   }
3192   else if(ansatz==LINEAR)
3193   {
3194     double* new_data=(double*)malloc(4*row_mesh.size()*4*col_mesh.size()*
3195                                      sizeof(double));
3196     A=Matrix(4*row_mesh.size(),4*col_mesh.size());
3197     for(int i=0;i<4*row_mesh.size();++i)
3198       for(int j=0;j<4*col_mesh.size();++j)
3199       {
3200         double value=compute_entry_DLP_linear(row_mesh,col_mesh,i,j);
3201         new_data[j*4*row_mesh.size()+i]=value;
3202       }
3203     A.data=new_data;
3204   }
3205   else if(ansatz==LINCONST)
3206   {
3207     double* new_data=(double*)malloc(4*row_mesh.size()*col_mesh.size()*
3208                                      sizeof(double));
3209     A=Matrix(row_mesh.size(),4*col_mesh.size());
3210     std::cout<<row_mesh.size()<<"␣Zeilen␣werden␣berechnet,␣derzeit␣Zeile:\n";
3211     for(int i=0;i<row_mesh.size();++i)
3212     {
3213       for(int j=0;j<4*col_mesh.size();++j)
3214       {
3215         double value=compute_DLP_RHS_constlin(row_mesh,col_mesh,i,j);
3216         new_data[j*row_mesh.size()+i]=value;
3217       }
3218     }
3219     A.data=new_data;
3220   }
3221   std::cout<<"\n\n";
3222 }
3223
3224 double* DLP_RHS::operator* (double* x)
3225 {
3226   double* b=MVM(A,x);
3227   return b;
3228 }
3229
3230
3231 }
```

Literaturverzeichnis

[1] H. W. Alt: *Lineare Funktionalanalysis*, Springer-Verlag, Berlin, Heidelberg, New York, 2002

[2] J. Bergh, J. Löfström: *Interpolation Spaces: An Introduction*, Springer-Verlag, Berlin, Heidelberg, New York, 1976

[3] D. Braess: *Finite elements: Theory, fast solvers, and applications in solid mechanics*, University Press, Cmbridge, 1997

[4] S. Börm, L. Grasedyck, W. Hackbusch: *Hierarchical Matrices*, Max-Planck-Institut fr Mathematik in den Naturwissenschaften, Leipzig, 2003

[5] C. Carstensen: *Efficiency of a posteriori BEM-error estimates for first-kind integral equations on quasi-uniform meshes*, Math. Comp. Volume 65, Number 213 (69-84), 1996

[6] C. Carstensen, B. Faermann: *Mathematical foundation of a posteriori error estimates and adaptive mesh-refining algorithms for boundary integral equations of the first kind*, Engineering Analysis with Boundary Elements 25 (497-509), 2001

[7] C. Carstensen, M. Maischak, E. P. Stephan: *A posteriori error estimate and h-adaptive algorithm on surfaces for Symm's integral equation*, Numer. Math. 90 (197-213), 2001

[8] C. Carstensen, D. Praetorius: *Averaging techniques for the effective numerical solution of Symm's integral equation of the first kind*, SIAM J. Sci. Comput. 27 (1226–1260), 2006

[9] C. Carstensen, E. P. Stephan: *A posteriori error estimates for boundary element methods*, Math. Comp. volume 64, number 210 (483-500), 1995

[10] W. Dahmen, R. Schneider: *Wavelets on manifolds. I: Construction and domain decomposition*, SIAM J. Math. Anal. 31 (184-230), 1999

[11] W. Dörfler, R. H. Nochetto: *Small data oscillation implies saturation assumption*, Numer. Math. 91 (1-12), 2002

[12] C. Erath, S. Ferraz-Leite, S. Funken, D. Praetorius: *Energy Norm based A Posteriori Error Estimation for Boundary Element Methods in two dimensions*, Preprint 07/2007, Institut für Analysis und Scientific Computing, TU Wien, Eingereicht bei Appl. Numer. Math.

[13] L. C. Evans: *Partial Differential Equations*, American Mathematical Society, Providence, Rhode Island, 1998

[14] B. Faermann: *Local a-posteriori error indicators for the Galerkin discretization of boundary integral equations*, Numer. Math. 79 (43-76), 1998

[15] S. Ferraz-Leite, S. Funken, D. Praetorius: *Averaging on large patches for integral equations in 3D*, work in progress 2007.

[16] S. Ferraz-Leite, D. Praetorius: *Simple A Posteriori Error Estimators for the h Version of the Boundary Element Method*, Preprint 01/2007, Institut für Analysis und Scientific Computing, TU Wien, Eingereicht bei Computing, 2007.

[17] L. Greengard, V. Rokhlin: *A new version of the Fast Multipole Method for Laplace equation in three dimensions.*, in Acta Numerica (229-269), Cambridge University Press, 1997

[18] I. G. Graham, L. Grasedyck, W. Hackbusch, S. A. Sauter: *Optimal Panel-Clustering in the Presence of Anisotropic Mesh Refinement*, Preprint, 2006

[19] I. G. Graham, W. Hackbusch, S. A. Sauter: *Finite elements on degenerate meshes: inverse-type inequalities and applications*, IMA J. Numer. Anal. 25 (379–407), 2005

[20] W. Hackbusch: *Integralgleichungen: Theorie und Numerik*, Teubner Verlag, Stuttgart, 1989 (Teubner-Studienbücher: Mathematik)

[21] W. Hackbusch: *A sparse matrix arithmetic based on $\mathcal{H}$-matrices. I: Introduction to $\mathcal{H}$-matrices.*, Computing 62 (89-108), 1999

[22] W. Hackbusch, Z. P. Nowak: *On the fast matrix multiplication in the boundary element method by panel clustering*, Numer. Math. 54 (463-491), 1989

[23] M. Maischak: *The analytical computation of the Galerkin elements for the Laplace, Lame and Helmholtz equation in 3D-BEM*, Preprint, Institut für Angewandte Mathematik, Universität Hannover, 2000

[24] D. Praetorius: *Integral Equations and Boundary Element Methods*, lecture notes, Institut für Analysis und Scientific Computing, TU Wien, 2007

[25] S. Sauter, C. Schwab: *Randelementmethoden: Analyse, Numerik und Implementierung schneller Algorithmen*, Teubner Verlag, Wiesbaden 2004

[26] O. Steinbach: *Adaptive boundary element methods based on computation schemes for Sobolev norms*, SIAM J. Sci. Comput. Vol. 22, No 2. (604-616), 2000

[27] O. Steinbach: *Numerische Näherungsverfahren für elliptische Randwertprobleme: Finite Elemente und Randelemente*, Teubner Verlag, Stuttgart, Leipzig, Wiesbaden, 2003

[28] J. Wloka: *Partielle Differentialgleichungen*, Teubner Verlag, 1964