TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

BACHELORARBEIT

Stabile Implementierung der Randelementmethode auf anisotrop verfeinerten Gittern in 3D

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

unter der Anleitung von
Ao.Univ.Prof. Dr. Dirk Praetorius
Dr. Michael Karkulik
Dipl.-Ing. Michael Feischl

durch
Peter Matthias Schaefer
e0825843@student.tuwien.ac.at
Volkertplatz 6/7
1020 Wien

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit beschäftigen wir uns mit der Randelementemethode für die homogene Laplace-Gleichung mit Dirichlet-Randbedingungen

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega \subset \mathbb{R}^3, \\ u &= g \quad \text{auf } \Gamma := \partial\Omega, \end{aligned} \tag{1.1}$$

wobei $\Delta u := \partial_x^2 u + \partial_y^2 u + \partial_z^2 u$ den Laplace-Operator bezeichnet und $\Omega \subset \mathbb{R}^3$ eine beschränkte Teilmenge von $\mathbb{R}^3$ mit Lipschitz-Rand Γ ist.
In Abschnitt 2 stellen wir die Randelementemethode für die homogene Laplace-Gleichung mit Dirichlet-Randbedingungen vor. Dabei verwenden wir den indirekten Ansatz, um anschließend mithilfe des Galerkin-Verfahrens die Gleichung zu lösen. An dieser Stelle werden wir auch kurz die Parametrisierung des Randes vorstellen. Wir werden im Folgenden den Rand in affine achsenorientierte Rechtecke T zerlegen, das heißt, die Punkte in einem Rechteck liegen in einer zu den Achsen des Koordinatensystems parallelen Ebene.
In Abschnitt 3 werden wir uns mit der approximativen Berechnung des Doppelintegrals

$$\int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \tag{1.2}$$

beschäftigen. $\kappa(\boldsymbol{x}, \boldsymbol{y})$ sei hierbei eine asymptotisch glatte Kernfunktion. Speziell interessieren wir uns hierbei für die asymptotisch glatte Funktion $\kappa(\boldsymbol{x}, \boldsymbol{y}) = |\boldsymbol{x} - \boldsymbol{y}|^{-1}$. Ziel wird es sein, das äußere Integral durch eine Gauss-Quadratur zu ersetzen. Unter bestimmten Zulässigkeitsbedingungen werden wir zeigen, dass der Quadraturfehler exponentiell schnell gegen 0 konvergiert. Weiterhin betrachten wir die in Abschnitt 2 auftretende Matrix $A \in \mathbb{R}^{n \times n}$, deren Einträge A_{jk} durch (1.2) bestimmt werden. Wir werden eine approximative Matrix A_p aufstellen, welche das durch Quadratur approximierte Integral für die zulässigen Einträge und für alle anderen das exakte Integral verwendet. Hiermit können wir dann zeigen, dass die approximative Matrix A_p in der Frobenius-Norm exponentiell schnell gegen A konvergiert.
In Abschnitt 4 fassen wir kurz zusammen, wie wir das Doppelintegral in einfache Integrale zerlegen und anschließend voll analytisch berechnen können. Hierzu werden wir uns weitgehend an [4] orientieren.
Abschließend werden wir kurz die numerische Umsetzung der Techniken vorstellen und anhand von numerischen Beispielen vergleichen. Hierbei wird uns die voll analytische und approximative Berechnung sowie die adaptive und uniforme Netzverfeinerung besonders interessieren.

2 Randelementemethode

An dieser Stelle betrachten wir die homogene Laplace-Gleichung mit Dirichlet-Randbedingungen

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega \subset \mathbb{R}^3, \\ u &= g \quad \text{auf } \Gamma := \partial\Omega, \end{aligned} \tag{2.1}$$

wobei $\Delta u := \partial_x^2 u + \partial_y^2 u + \partial_z^2 u$ den Laplace-Operator bezeichnet und Ω eine beschränkte Teilmenge von $\mathbb{R}^3$ mit Lipschitz-Rand ist.

2.1 Galerkin-Verfahren

Die Fundamentallösung des Laplace-Operators [7, Kapitel 5.1] ist für $\Omega \subset \mathbb{R}^3$ gegeben durch

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

Weiterhin ist das Einfachschichtpotential $\tilde{V}$ gegeben durch

$$\tilde{V}\phi(\boldsymbol{x}) := \int_\Gamma G(\boldsymbol{x}-\boldsymbol{y})\phi(\boldsymbol{y}) ds_{\boldsymbol{y}} \quad \text{für } \boldsymbol{x} \in \Omega.$$

Das Integral ist hier als Oberflächenintegral zu verstehen. Mit γ_0 bezeichnen wir den Spuroperator, der für $C^\infty(\bar{\Omega})$ Funktionen über

$$\gamma_0 u = u|_\Gamma$$

definiert ist und durch stetige Fortsetzung zu einem Operator $\gamma_0 \in L(H^1(\Omega)), H^{1/2}(\Gamma))$ wird. In der schwachen Formulierung lautet die Laplace-Gleichung

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } H^{-1}(\Omega), \\ \gamma_0 u &= g \quad \text{auf } H^{1/2}(\Gamma). \end{aligned}$$

Gemäß [7, Kapitel 6.2 und 6.3] kann $\tilde{V}$ auch als Operator $\tilde{V} \in L(H^{-1/2}(\Gamma), H^1(\Omega))$ aufgefasst werden und es gilt

$$-\Delta\tilde{V}\phi = 0 \in H^{-1}(\Omega) \quad \text{für alle } \phi \in H^{-1/2}(\Gamma)$$

Insbesondere ist auch die Spur

$$\gamma_0\tilde{V} \in L(H^{-1/2}(\Gamma), H^{1/2}(\Gamma))$$

wohldefiniert. Wir machen nun einen indirekten Ansatz $u = \tilde{V}\phi$, wodurch

$$V\phi = g \tag{2.2}$$

mit $V := \gamma_0\tilde{V}$ gilt. Ziel ist es nun, aus (2.2) eine Funktion ϕ zu bestimmen, die die obige Gleichung erfüllt. Dann ist $\tilde{V}\phi$ die Lösung des Problems (2.1).
Da wir das Problem (2.2) im Allgemeinen nicht exakt lösen können, werden wir es mithilfe des Galerkin-Verfahrens näherungsweise lösen. Die Idee dabei ist, $H^{-1/2}(\Gamma)$ durch einen endlich dimensionalen Unterraum zu ersetzen.

Bezeichne nun $\langle \cdot, \cdot \rangle$ das erweiterte L^2-Skalarprodukt, so existiert, da das Einfachschichtpotential V ein symmetrischer und elliptischer Isomorphismus ist, auf $\widetilde{H}^{-1/2}$ ein äquivalentes Skalarprodukt $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ mit $\langle\!\langle \phi, \psi \rangle\!\rangle := \langle V\phi, \psi \rangle$ und der induzierten Norm $|\!|\!| \cdot |\!|\!|$. Sei nun ϕ die eindeutige Lösung von (2.2) und bezeichne g die Dirichlet-Daten am Rand. Dann gilt

$$\langle\!\langle \phi, \psi \rangle\!\rangle = \langle g, \psi \rangle \quad \text{für alle } \psi \in \widetilde{H}^{-1/2}. \tag{2.3}$$

Sei nun $\mathcal{T}_\ell = \{T_1, T_2, \ldots, T_N\}$ eine Partition des Randes Γ in N Randstücke mit den charakteristischen Funktionen

$$\chi_i(\boldsymbol{x}) = \begin{cases} 1 & \text{für alle } \boldsymbol{x} \in T_i \\ 0 & \text{sonst} \end{cases} \quad \text{für alle } i \in \{1, 2, \ldots, N\}$$

und sei $P^0(\mathcal{T}_\ell) := span\{\chi_i : i = 1, \ldots, N\}$ der aufgespannte Teilraum von $\tilde{H}^{-1/2}(\Gamma)$. Wir suchen also die Lösung $\phi_\ell \in P^0(\mathcal{T}_\ell)$ für

$$\langle\!\langle \phi_\ell, \psi_\ell \rangle\!\rangle = \langle f, \psi_\ell \rangle \quad \text{für alle } \psi_\ell \in P^0(\mathcal{T}_\ell),$$

was aufgrund der Basiseigenschaft von χ_i und dem Lemma von Lax-Milgram eindeutig und äquivalent ist zu

$$\langle\!\langle \phi_\ell, \chi_j \rangle\!\rangle = \langle f, \chi_j \rangle \quad \text{für alle } j \in \{1, 2, \ldots, N\}. \tag{2.4}$$

Dadurch können wir das Problem als Summe anschreiben

$$\sum_{j=1}^{N} x_{\ell,j} \langle\!\langle \chi_j, \chi_i \rangle\!\rangle = \langle g, \chi_i \rangle,$$

wobei $\boldsymbol{x}_\ell = \{x_{\ell,1}, x_{\ell,2}, \ldots, x_{\ell,N}\}$ der Koordinatenvektor von ϕ_ℓ zur Basis $\chi_1, \ldots, \chi_N$ ist. So schreiben wir

$$\boldsymbol{V}\boldsymbol{x}_\ell = \boldsymbol{g}_\ell, \tag{2.5}$$

mit $\boldsymbol{V} = (\langle\!\langle \chi_j, \chi_i \rangle\!\rangle)_{ij}$ und $\boldsymbol{g}_\ell = (\langle g, \chi_j \rangle)_j$. In Integralschreibweise können wir $\boldsymbol{V}$ anschreiben als

$$\boldsymbol{V}_{ij} := \frac{1}{4\pi} \int_{T_j} \int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}. \tag{2.6}$$

2.2 Netze

Für die Diskretisierung des Problems wollen wir nun einige Begriffe definieren. Zunächst wollen wir ein achsenorientiertes Rechteck beschreiben, dessen Seiten parallel zu den Achsen des kartesischen Koordinatensystems liegen.

Definition 2.1 *Sei $\boldsymbol{v} \in \mathbb{R}^3$, $\boldsymbol{a}, \boldsymbol{b} \in \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T\}$ mit $\boldsymbol{a} \neq \boldsymbol{b}$ und $a, b \in \mathbb{R}$ mit $a, b > 0$. Dann nennen wir*

$$T := \{\boldsymbol{v} + \lambda_1 a\boldsymbol{a} + \lambda_2 b\boldsymbol{b} \mid \lambda_1, \lambda_2 \in [0,1]\}$$

achsenorientiertes Rechteck. Ferner sei

$$\gamma_T := [0,1]^2 \to T : \lambda_1, \lambda_2 \mapsto \boldsymbol{v} + \lambda_1 a\boldsymbol{a} + \lambda_2 b\boldsymbol{b}$$

die zu T zugehörige Parametrisierung.

Bemerkung 2.2 *Weiterhin werden wir die vier Eckpunkte des achsenorientierten Rechtecks, beginnend in* $\boldsymbol{v}$*, mit* $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_4$*, also* $\boldsymbol{v} = \boldsymbol{k}_1$*, bezeichnen, wobei die Menge aller Knoten des Rechtecks* $\mathcal{K}_T$ *sei. Die Reihenfolge der Knoten sei dabei so gewählt, dass der Normalenvektor* $\boldsymbol{n} = \overline{\boldsymbol{k}_1\boldsymbol{k}_2} \times \overline{\boldsymbol{k}_1\boldsymbol{k}_4}$ *nach außen zeigt. Außerdem benennen wir die Menge der Kanten mit* $\mathcal{E}_T$*, bestehend aus den vier Kanten* $e_1, \ldots, e_4$*. In Abbildung 1 wurde ein Rechteck mit den Bezeichnungen kurz skizziert.*

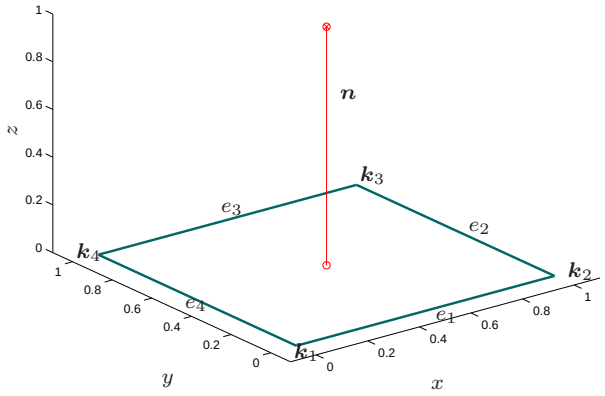


Abbildung 1: Achsenorientiertes Rechteck mit Beschriftung der Eckpunkte $\{\boldsymbol{k}_1, \ldots \boldsymbol{k}_4\}$, der Kanten $\{e_1, \ldots e_4\}$ und dem Normalenvektor $\boldsymbol{n}$ entsprechend der Bemerkung 2.2.

Des Weiteren werden wir für die Berechnungen noch Aussagen über die Größe eines Elements sowie über den Abstand zweier Elemente festhalten.

Definition 2.3 *Sei* $a, b \in \mathbb{R}$ *für* T *definiert wie in Definition 2.1, dann heißt*

$$\mathcal{D}(T) = (a^2 + b^2)^{1/2}$$

Durchmesser von T*. Weiterhin bezeichne*

$$\mathcal{D}_{\boldsymbol{a}}(T) = a$$

die Seitenlänge des Rechtecks T *in Richtung* $\boldsymbol{a}$*. Die Seitenlänge in Richtung* $\boldsymbol{b}$ *definieren wir analog.*

Definition 2.4 *Der Abstand zweier Rechtecke* T_j *und* T_k *sei definiert durch*

$$\operatorname{dist}(T_j, T_k) = \min\{|\boldsymbol{x} - \boldsymbol{y}| \mid \boldsymbol{x} \in T_j, \boldsymbol{y} \in T_k\},$$

wobei das Minimum aufgrund der Kompaktheit von T_j, T_k *angenommen wird. Weiterhin definieren wir auch den Abstand in einer bestimmten Richtung* $\boldsymbol{a}$ *durch*

$$\operatorname{dist}_{\boldsymbol{a}}(T_j, T_k) = \min\{|\boldsymbol{a}^T \cdot (\boldsymbol{x} - \boldsymbol{y})| \mid \boldsymbol{x} \in T_j, \boldsymbol{y} \in T_k\}$$

mit dem Skalarprodukt $(\cdot)$ *und* $\boldsymbol{a} \in \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T\}$.

Mit diesen Vorüberlegungen definieren wir uns die Diskretisierung des Randes Γ.

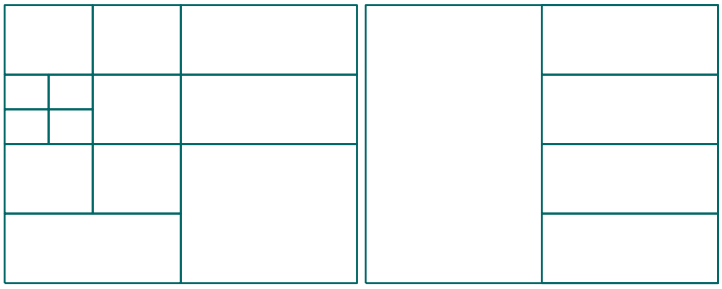
Definition 2.5 *Sei* $\mathcal{T}_\ell = \{T_1, T_2, \ldots, T_N\}$ *eine endliche Menge von achsenorientierten Rechtecken. Es bezeichne* $\mathcal{K}_\ell := \bigcup_{T \in \mathcal{T}} \mathcal{K}_T$ *die Menge aller Knoten von* $\mathcal{T}_\ell$ *und* $\mathcal{E}_\ell := \bigcup_{T \in \mathcal{T}} \mathcal{E}_T$ *die Menge aller Kanten. Wir nennen* $\mathcal{T}_\ell$ *eine Partition von* Γ*, falls*

- $\overline{\Gamma} = \bigcup_{j=1}^{N} T_j$
- $|T_j \cap T_k| = 0$ *für den Schnitt zweier Elemente* $T_j, T_k \in \mathcal{T}_\ell$ *mit* $T_j \neq T_k$.

Hier bezeichne $|\cdot|$ *das 2-dimensionale Oberflächenmaß. Weiterhin sei der Schnitt* $T_j \cap T_k$ *zweier Elemente* $T_j, T_k \in \mathcal{T}_\ell$ *mit* $T_j \neq T_k$

- *leer,*
- *oder ein gemeinsamer Knoten von* T_j *und* T_k*,*
- *oder eine (nicht zwingend gemeinsame) Kante von* T_j *oder* T_k.

Ferner liegen auf jeder Kante von $\mathcal{T}_\ell$ *maximal 3 Knoten.*



(a) gültige Partition

(b) Ungültige Partition, da mehr als 3 Knoten auf einer Kante liegen.

Abbildung 2: Beispiel Partitionen zur Definition 2.5

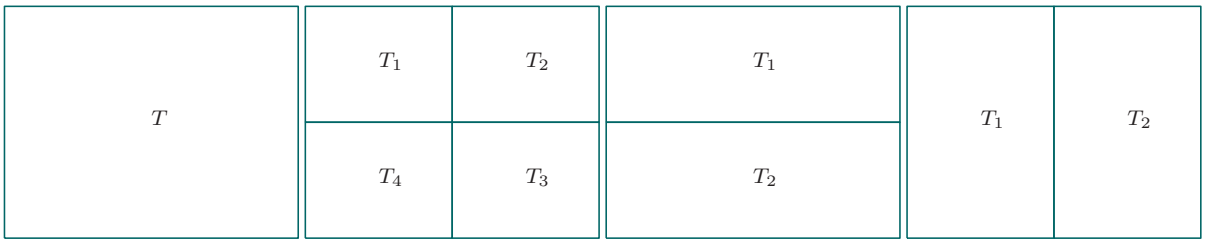
2.3 Verfeinern

Definition 2.6 (Lokale Verfeinerung) *Ein Element* $T \in \mathcal{T}$ *wird isotrop in vier Elemente* $T_1, \ldots, T_4$ *geteilt, wenn* $T = \bigcup_{n=1}^{4} T_n$ *gilt und alle Söhne* $T_1, \ldots, T_4$ *ähnlich sind zum Vaterelement* T*. Weiterhin wird ein Element* $T \in \mathcal{T}$ *anisotrop in zwei Elemente* T_1, T_2 *geteilt, wenn ebenfalls* $T = T_1 \cup T_2$ *gilt und* T_1, T_2 *untereinander ähnlich sind. Hierbei kann* T *entweder horizontal oder vertikal geteilt werden, wie in Abbildung 3 gezeigt ist.*

Definition 2.7 *Für eine Partition* $\mathcal{T}_\ell$ *bezeichnen wir mit* $\widehat{\mathcal{T}_\ell}$ *jene Partition die entsteht, wenn alle Elemente isotrop verfeinert werden.*

Für das Verständnis des folgenden Algorithmus benötigen wir noch einige Beobachtungen:

- Sind $e, \tilde{e} \in \mathcal{E}_T$ Kanten von $T \in \mathcal{T}_\ell$ mit $e \cap \tilde{e} = \emptyset$, so liegen diese gegenüber und haben insbesondere dieselbe Länge. Falls e verfeinert werden soll, muss zwingend auch $\tilde{e}$ verfeinert werden.
- Es bezeichnet $\mathcal{S}_\ell := \{(e, T) \mid T \in \mathcal{T}_\ell, e \in \mathcal{E}_T\}$ die Menge aller Elemente mit zugehörigen Kanten. Hierbei kann es aufgrund der maximal drei Knoten auf einer Kante vorkommen, dass eine Kante $e \in \mathcal{E}_\ell$, die im Inneren von Γ liegt, nur zu einem Element $T \in \mathcal{T}_\ell$ gehört und nicht zu zwei. In diesem Fall gibt es ein Element $\hat{T} \in \mathcal{T}_\ell$ mit Kante $\hat{e} \in \mathcal{E}_\ell$ in der e ganz enthalten ist. Sollte (e, T) verfeinert werden, so muss zwingend auch $(\hat{e}, \hat{T})$ verfeinert werden, damit nicht mehr als drei Knoten auf der Kante $\hat{e}$ entstehen.



(a) markiertes Element (b) isotrope Teilung (c) vertikale Teilung (d) horizontal Teilung

Abbildung 3: Entsprechend der Definition 2.6 wird ein markiertes Element durch isotrope Teilung in vier ähnliche Elemente geteilt. Durch vertikale Teilung werden die beiden vertikalen Kanten halbiert, wodurch zwei übereinander liegende ähnliche Elemente entstehen. Die beiden horizontalen Kanten werden durch horizontale Teilung halbiert, hierbei entstehen zwei ähnliche nebeneinander liegende Elemente.

Algorithmus 2.8 (Verfeinern) *Sei $\mathcal{T}_\ell$ eine Partition und $\sqcap_\ell \subseteq \mathcal{S}_\ell$ eine Menge markierter Kanten. Nun sei $\sqcap_\ell^{(0)} := \sqcap_\ell$ und $i = 0$. Dann gehe so vor:*

(i). $\sqcap_\ell^{(i+1/2)} := \sqcap_\ell^{(i)} \cup \{(e,T) \in \mathcal{S}_\ell \setminus \sqcap_\ell^{(i)} \mid \exists (\hat{e},\hat{T}) \in \sqcap_\ell^{(i)} \text{ mit } e \supsetneqq \hat{e}\}$

(ii). $\sqcap_\ell^{(i+1)} := \sqcap_\ell^{(i+1/2)} \cup \{(e,T) \in \mathcal{S}_\ell \setminus \sqcap_\ell^{(i+1/2)} \mid \exists (\tilde{e},\tilde{T}) \in \sqcap_\ell^{(i+1/2)} \text{ mit } T = \tilde{T} \text{ und } e \cap \tilde{e} = \emptyset\}$

(iii). Falls $\sqcap_\ell^{(i)} \subsetneqq \sqcap_\ell^{(i+1)}$*, erhöhe Zähler* $i \mapsto i+1$ *und gehe zu Schritt (i)*

(iv). Teile alle Elemente aus $\mathcal{T}_\ell$ bezüglich der markierten Kanten $\sqcap_\ell^{(i)}$, Definition 2.6 erfüllend.

3 Analytische und Semi-analytische Berechnung

Ziel dieses Abschnitts ist die approximative Berechnung des Integrals

$$A_{jk} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}, \tag{3.1}$$

auf achsenorientierten Rechtecken $T_j, T_k \subset \mathbb{R}^3$, beziehungsweise als Spezialfall davon die Berechnung des Integrals

$$A_{jk} = \int_{T_j} \int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}, \tag{3.2}$$

unter bestimmten Voraussetzungen an die affinen Randstücke T_j, T_k und den asymptotisch glatten Integranden $\kappa : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$.

3.1 Interpolation

An dieser Stelle werden wir zunächst den Interpolationsoperator auf dem Intervall $[0, 1]$ definieren. Ferner wollen wir mithilfe von Chebyshev-Knoten eine Fehlerabschätzung für die Chebyshev'sche Interpolation auf d-dimensionalen Würfeln $[0, 1]^d$ mit $d \in \mathbb{N}$ definieren. Im Folgenden bezeichnet $\mathcal{P}^p$ die Menge aller Polynome vom Grad höchstens p auf $[0, 1]$.

Definition 3.1 *Für einen festen Grad $p \in \mathbb{N}$ und paarweise verschiedene Knoten $x_j \in [0, 1]$ lautet das Lagrange'sche Interpolationsproblem: Zu gegebenen Funktionswerten $y_j \in \mathbb{R}$ finde ein Polynom $q \in \mathcal{P}^p$ so, dass*

$$q(x_j) = y_j \quad \text{für alle } j = 0, \ldots, p.$$

Wir wissen aus [5, Theorem 1.6], dass für das Lagrange'sche Interpolationsproblem die eindeutige Lösung gegeben ist durch

$$q = \sum_{j=0}^{p} y_j L_j,$$

wobei die Lagrange-Polynome L_j definiert sind durch

$$L_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{p} \frac{x - x_i}{x_j - x_i}.$$

Mit diesem Wissen definieren wir uns nun den Interpolationsoperator $\mathcal{I}_p : \mathcal{C}[0, 1] \to \mathcal{P}^p$

$$\mathcal{I}_p u := \sum_{j=0}^{p} u(x_j) L_j.$$

Wie an der Fehlerabschätzung aus [5, Theorem 1.17] für den Interpolationsoperator

$$\|u - \mathcal{I}_p u\|_{\infty,[0,1]} \leq \frac{\|u^{(p+1)}\|_{\infty,[0,1]}}{(p+1)!} \max_{x \in [0,1]} \prod_{j=0}^{p} |x - x_j| \quad \text{für alle } u \in \mathcal{C}^{p+1}([0, 1])$$

zu sehen ist, hängt diese noch von der Wahl der Knoten ab. Um den Fehler weiter zu verbessern, werden wir das Produkt

$$\max_{x\in[0,1]} \prod_{j=0}^{p} |x - x_j|$$

durch benutzen von Chebyshev-Knoten minimieren [5, Definition 1.22]. Diese sind für das Intervall $[-1, 1]$ gegeben durch

$$x_j = \cos\left(\frac{2j+1}{p+1}\frac{\pi}{2}\right) \quad \text{für } j = 0, \ldots, p.$$

Durch affine Transformation der Chebyshev-Knoten [5, Theorem 1.25] ergibt sich die Fehlerabschätzung

$$\|u - \mathcal{I}_p u\|_{\infty,[0,1]} \leq \frac{4^{-p}}{(p+1)!} \|u^{(p+1)}\|_{\infty,[0,1]} \quad \text{für alle } u \in \mathcal{C}^{p+1}([0,1])$$

für Chebyshev-Polynome.
Ferner wollen wir nun die Interpolation auf Boxen der Gestalt $[0, 1]^d$ mit $d \in \mathbb{N}$ einführen. Hierzu definieren wir uns mit dem Tensorprodukt den Interpolationsoperator

$$\mathcal{I}_p^d := \bigotimes_{i=1}^{d} \mathcal{I}_p \quad \text{für Boxen } [0,1]^d.$$

Weiterhin benötigen wir für die Abschätzung die Lebesgue-Konstante

$$\Lambda_p := \max_{x\in[0,1]} \sum_{j=0}^{p} |L_j(x)|.$$

Dann können wir die Fehlerabschätzung für den Interpolationsoperator $\mathcal{I}_p^d$ aus [6, Theorem 2.12] anschreiben

$$\|u - \mathcal{I}_p^d u\|_{\infty,[0,1]^d} \leq \frac{4^{-p}\Lambda_p^{d-1}}{(p+1)!} \sqrt{d}^{(p+1)} \sum_{j=1}^{d} \|\partial_j^{(p+1)} u\|_{\infty,[0,1]^d} \quad \text{für alle } u \in \mathcal{C}^{p+1}([0,1]^d).$$

3.2 Gauss-Quadratur

Im Folgenden wollen wir die klassische Gauss-Quadratur definieren. Unter einer Quadratur verstehen wir die approximative Berechnung eines Integrals der Form

$$\int_0^1 f(x)dx$$

durch die Summe

$$\mathcal{Q}_n(f) := \sum_{k=0}^{n} w_k f(t_k).$$

Die t_k sind hierbei die Knoten, die w_k Gewichte. Eine Quadratur heißt exakt für eine Funktion f, falls $\mathcal{Q}_n(f) = \int_0^1 f(x)dx$ ist.
Weiterhin hat eine Quadratur den Exaktheitsgrad m, wenn sie für alle Polynome bis zum Grad

m exakt ist.
Eine Quadratur heißt interpolatorisch (vom Grad n), falls für jede integrierbare, auf $(0,1)$ stetige Funktion f

$$\mathcal{Q}_n(f) := \sum_{k=0}^{n} w_k f(t_k) = \sum_{k=0}^{n} w_k p(t_k) = \int_0^1 p(x)dx$$

gilt, wobei p das Interpolationspolynom von f vom Grad n zu den Knoten $t_0, \ldots, t_n$ ist.

3.3 Volle Quadratur

Zunächst wollen wir zeigen, dass wir die vier Integrale aus (3.2) gut durch die Gauss-Quadratur approximieren können. Hierzu werden wir einige Vorüberlegungen benötigen.

Definition 3.2 *Die Kern-Funktion $\kappa(\boldsymbol{x}, \boldsymbol{y})$ heißt asymptotisch glatt, falls sie glatt ist für $\boldsymbol{x} \neq \boldsymbol{y}$ und Konstanten $c_1, c_2 > 0$ und eine Ordnung der Singularität $s \in \mathbb{R}$ existieren, sodass*

$$|\partial_{\boldsymbol{x}}^{\alpha} \partial_{\boldsymbol{y}}^{\beta} \kappa(\boldsymbol{x}, \boldsymbol{y})| \leq c_1 (c_2 (|\boldsymbol{x} - \boldsymbol{y}|)^{-(|\alpha|+|\beta|+s)} (|\alpha| + |\beta|)!$$

für alle Multiindizes $\alpha, \beta \in \mathbb{N}_0^d$ mit $|\alpha| + |\beta| \geq 1$ gilt.

Wie wir im Folgenden sehen werden, lassen sich asymptotisch glatte Kernfunktionen außerhalb des Singularitätsbereichs gut durch Polynome interpolieren. Hierzu benötigen wir aus [1, Theorem 3.2] das folgende Lemma:

Lemma 3.3 *Die Funktion $u \in \mathcal{C}^{\infty}([0,1]^d)$ erfülle für Konstanten $C_u, \rho_u > 0$*

$$\|\partial_j^n u\|_{\infty,[0,1]^d} \leq C_u \rho_u^n n! \quad \text{für alle } j \in \{1, \ldots, d\} \text{ und } n \in \mathbb{N}_0.$$

Dann gilt für alle $p \in \mathbb{N}_0$

$$\|u - \mathcal{I}_p^d u\|_{\infty,[0,1]^d} \leq C_u 8e(1 + \rho_u \sqrt{d}) \Lambda_p^d (p+1) \left(1 + \frac{2}{\rho_u \sqrt{d}}\right)^{-(p+1)}. \tag{3.3}$$

Weiterhin wollen wir uns kurz die Ableitung der asymptotisch glatten Kernfunktion mit einer Parametrisierung anschauen.

Lemma 3.4 *Sei $\kappa(\cdot,\cdot) : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ eine asymptotisch glatte Kernfunktion und sei weiterhin $g : \mathbb{R}^4 \to \mathbb{R}^6$ die Parametrisierung zweier achsenorientierten Rechtecke T_j und T_k, mit $T_j \cap T_k = \emptyset$ sowie*

$$g(\lambda) = (\gamma_j(\lambda_1, \lambda_2), \gamma_k(\lambda_3, \lambda_4)), \tag{3.4}$$

wobei γ_j, γ_k die Parametrisierungen aus Definition 2.1 seien.
Mit der Hilfsfunktion $t_{jk} : \mathbb{N}^4 \to \mathbb{N}^6$

$$t_{jk}(\alpha) = \begin{pmatrix} \alpha_1 \cdot \boldsymbol{a} + \alpha_2 \cdot \boldsymbol{b} \\ \alpha_3 \cdot \tilde{\boldsymbol{a}} + \alpha_4 \cdot \tilde{\boldsymbol{b}} \end{pmatrix}, \tag{3.5}$$

wobei $\boldsymbol{a}, \boldsymbol{b}, \tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}} \in \mathbb{R}^3$ die paarweise verschiedenen Einheitsvektoren zu Parametrisierungen γ_j, γ_k sind, gilt dann für die Verknüpfung $\kappa \circ g$ die Kettenregel

$$|\partial^{\alpha}(\kappa \circ g)(\lambda)| = \mathcal{D}_{\boldsymbol{a}}(T_j)^{\alpha_1} \mathcal{D}_{\boldsymbol{b}}(T_j)^{\alpha_2} \mathcal{D}_{\tilde{\boldsymbol{a}}}(T_k)^{\alpha_3} \mathcal{D}_{\tilde{\boldsymbol{b}}}(T_k)^{\alpha_4} |\partial^{t_{jk}(\alpha)} \kappa(g(\lambda))|$$

für jeden Multiindex $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ mit $|\alpha| \geq 1$.

Beweis. Die Ableitung der Funktion $\kappa \circ g : \mathbb{R}^4 \to \mathbb{R}^6 \to \mathbb{R}$ ist

$$\partial(\kappa \circ g)(\lambda) = \partial\kappa(g(\lambda)) \circ \partial g(\lambda) \quad \in \mathbb{R}^{1\times 4}.$$

Mithilfe der Jacobimatrizen $A := \partial\kappa(g(\lambda)) \in \mathbb{R}^{1\times 6}$, $B := \partial g(\lambda) \in \mathbb{R}^{6\times 4}$ untersuchen wir zunächst die partiellen Ableitungen

$$\partial_m(\kappa \circ g)(\lambda) = (AB)_{1m} = \sum_{\ell=1}^{6} A_{1\ell}B_{\ell m} = \sum_{\ell=1}^{6} \partial_\ell\kappa(g(\lambda))\partial_m g_\ell(\lambda). \tag{3.6}$$

Hierzu treffen wir einige Vorüberlegungen.
Für die ersten drei Komponenten von g gilt

$$g_{1,2,3}(\lambda) = \gamma_j(\lambda_1, \lambda_2) = \boldsymbol{v} + \lambda_1 a\boldsymbol{a} + \lambda_2 b\boldsymbol{b},$$

mit den Vektoren $\boldsymbol{a}, \boldsymbol{b} \in \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T\}$, $\boldsymbol{a} \neq \boldsymbol{b}$ und Skalaren $a, b \in \mathbb{R}$ mit $a, b > 0$ der Parametrisierung zum Rechteck T_j, wodurch sich die folgenden Ableitungen ergeben.

$$\partial_1 g_{1,2,3}(\lambda) = a\boldsymbol{a} \qquad \partial_2 g_{1,2,3}(\lambda) = b\boldsymbol{b} \qquad \partial_{3,4} g_{1,2,3}(\lambda) = 0$$

Für die letzten drei Komponenten, wobei für die Parametrisierung des Rechtecks T_k die Variablen $\tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}$ analog definiert seien,

$$g_{4,5,6}(\lambda) = \gamma_k(\lambda_3, \lambda_4) = \tilde{\boldsymbol{v}} + \lambda_3\tilde{a}\tilde{\boldsymbol{a}} + \lambda_4\tilde{b}\tilde{\boldsymbol{b}}$$

können wir die Ableitungen ebenfalls anschreiben. Hierbei müssen wir nur beachten, dass sich die Indizierung von $\tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}$ ändert, weshalb wir hier die Komponenten

$$\partial_{1,2} g_\ell(\lambda) = 0 \qquad \partial_3 g_\ell(\lambda) = \tilde{a}\tilde{\boldsymbol{a}}_{\ell-3} \qquad \partial_4 g_\ell(\lambda) = \tilde{b}\tilde{\boldsymbol{b}}_{\ell-3}$$

für $\ell \in \{4,5,6\}$ erhalten.
Mit diesen Vorüberlegungen erhalten wir, da κ asymptotisch glatt ist, den Gradient $AB \in \mathbb{R}^{1\times 4}$

$$AB = \begin{pmatrix}\partial_1\kappa(g(\lambda)) & \cdots & \partial_6\kappa(g(\lambda))\end{pmatrix} \cdot \begin{pmatrix} a\boldsymbol{a} & b\boldsymbol{b} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \tilde{a}\tilde{\boldsymbol{a}} & \tilde{b}\tilde{\boldsymbol{b}} \end{pmatrix}. \tag{3.7}$$

Wenn wir die Summen der Einträge $(AB)_{1m}$ genauer betrachten, sehen wir, dass jeweils nur ein Eintrag $\neq 0$ ist, da $\boldsymbol{a}, \boldsymbol{b}, \tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}$ Einheitsvektoren sind.
Mit der Hilfsfunktion

$$\begin{aligned} ind : \{(1,0,0)^T, (0,1,0)^T, (0,0,1)^T\} &\to \{1,2,3\} \\ \boldsymbol{x} &\mapsto (1,2,3) \cdot \boldsymbol{x} \end{aligned}$$

für die Einheitsvektoren mit dem Skalarprodukt $(\cdot)$ vereinfachen wir den Gradient zu

$$\begin{aligned} \partial_1(\kappa \circ g)(\lambda) &= a\partial_{ind(\boldsymbol{a})}\kappa(g(\lambda)) & \partial_3(\kappa \circ g)(\lambda) &= \tilde{a}\partial_{ind(\tilde{\boldsymbol{a}})+3}\kappa(g(\lambda)) \\ \partial_2(\kappa \circ g)(\lambda) &= b\partial_{ind(\boldsymbol{b})}\kappa(g(\lambda)) & \partial_4(\kappa \circ g)(\lambda) &= \tilde{b}\partial_{ind(\tilde{\boldsymbol{b}})+3}\kappa(g(\lambda)), \end{aligned}$$

wobei auch hier wieder auf die richtige Indizierung geachtet werden muss.
Des Weiteren bestehen die ersten Ableitungen nur aus einem Skalar und dem asymptotisch glatten Kern $\partial_\ell\kappa(g(\lambda))$, weshalb wir die Kettenregel beliebig oft anwenden können. Mithilfe der Funktion $t_{jk} : \mathbb{N}^4 \to \mathbb{N}^6$ erhalten wir abschließend für einen Multiindex $\alpha \in \mathbb{N}_0^4$ die partiellen Ableitungen

$$\partial^\alpha(\kappa \circ g)(\lambda) = a^{\alpha_1} b^{\alpha_2} \tilde{a}^{\alpha_3} \tilde{b}^{\alpha_4} \partial^{t_{jk}(\alpha)}\kappa(g(\lambda)).$$

□

Für die Interpolation werden wir folgende Bedingung verwenden.

Definition 3.5 *Seien $T_j, T_k \in \mathbb{R}^3$ achsenorientierte Rechtecke und sei $\zeta_Q > 0$ fest. Dann heißt das Paar (T_j, T_k) ζ_Q-zulässig genau dann, wenn*

$$\operatorname{dist}(T_j, T_k) \geq \zeta_Q \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}. \tag{3.8}$$

Andernfalls heißt das Paar (T_j, T_k) ζ_Q-unzulässig.

Mit diesen Vorüberlegungen zeigen wir nun den folgenden Satz über die Interpolation der asymptotisch glatten Kernfunktion unter der Parametrisierung.

Satz 3.6 *Seien $T_j, T_k \subseteq \mathbb{R}^3$ zwei ζ_Q-zulässige Rechtecke. Sei $\kappa(\cdot,\cdot)$ eine asymptotisch glatte Kernfunktion mit Konstanten $c_1, c_2 > 0$ und Singularitätsordung $s \geq 0$. Dann gilt mit*

$$C_{\zeta_Q,j,k} := 8e \frac{c_1}{(c_2 \operatorname{dist}(T_j, T_k))^s} \left(1 + \frac{2}{c_2 \zeta_Q}\right)$$

die Abschätzung

$$\|\kappa(\gamma_j(\cdot), \gamma_k(\cdot)) - \mathcal{I}_p^4 \kappa(\gamma_j(\cdot), \gamma_k(\cdot))\|_{\infty,[0,1]^4} \leq C_{\zeta_Q,j,k} \Lambda_p^4 (p+1) \left(1 + c_2 \zeta_Q\right)^{-(p+1)}.$$

Beweis. Zunächst definieren wir die Konstanten

$$C_{\kappa,j,k} = \frac{c_1}{(c_2 \operatorname{dist}(T_j, T_k))^s} \quad \text{und} \quad \rho_\kappa = \frac{1}{c_2 \zeta_Q}$$

und können dann die Konstante $C_{\zeta_Q,j,k}$ kurz

$$C_{\zeta_Q,j,k} = 8e C_{\kappa,j,k}(1 + 2\rho_\kappa)$$

schreiben.
Bezeichne $g_{jk} : [0,1]^4 \to (T_j \times T_k) \subset \mathbb{R}^6$ die Parametrisierung

$$g_{jk}(\lambda) = (\gamma_j(\lambda_1, \lambda_2), \gamma_k(\lambda_3, \lambda_4)) \tag{3.9}$$

mit der jeweiligen Parametrisierung γ_j, γ_k zu T_j, T_k aus Definition 2.1. So gilt aufgrund von Lemma 3.4

$$\begin{aligned}
|\partial_\lambda^\alpha \kappa(\gamma_j(\lambda_1, \lambda_2), \gamma_k(\lambda_3, \lambda_4))| &= |\partial_\lambda^\alpha \kappa(g_{jk}(\lambda))| \\
&= \mathcal{D}_{\boldsymbol{a}}(T_j)^{\alpha_1} \mathcal{D}_{\boldsymbol{b}}(T_j)^{\alpha_2} \mathcal{D}_{\tilde{\boldsymbol{a}}}(T_k)^{\alpha_3} \mathcal{D}_{\tilde{\boldsymbol{b}}}(T_k)^{\alpha_4} \left|\partial^{t_{jk}(\alpha)} \kappa(g_{jk}(\lambda))\right| \\
&\leq \mathcal{D}(T_j)^{\alpha_1+\alpha_2} \mathcal{D}(T_k)^{\alpha_3+\alpha_4} \left|\partial^{t_{jk}(\alpha)} \kappa(g_{jk}(\lambda))\right| \\
&\leq \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{|\alpha|} \left|\partial^{t_{jk}(\alpha)} \kappa(g_{jk}(\lambda))\right|,
\end{aligned}$$

mit Multiindex $\alpha = (\alpha_1, \ldots, \alpha_4)$, wobei $|\alpha| > 0$ sei. Ferner gilt mit Definition 3.2 und den Konstanten $C_{\kappa,j,k}, \rho_\kappa$ die Abschätzung

$$\begin{aligned}
&\max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{|\alpha|} \left|\partial^{t_{jk}(\alpha)} \kappa(g_{jk}(\lambda))\right| \\
&\quad \leq \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{|\alpha|} c_1 (c_2 |\gamma_j(\lambda_1, \lambda_2) - \gamma_k(\lambda_3, \lambda_4)|)^{-(|\alpha|+s)} |\alpha|! \\
&\quad \leq \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{|\alpha|} c_1 (c_2 \operatorname{dist}(T_j, T_k))^{-(|\alpha|+s)} |\alpha|! \\
&\quad = \frac{c_1}{(c_2 \operatorname{dist}(T_j, T_k))^s} \left(\frac{\max(\mathcal{D}(T_j), \mathcal{D}(T_k))}{c_2 \operatorname{dist}(T_j, T_k)}\right)^{|\alpha|} |\alpha|! \\
&\quad \leq C_{\kappa,j,k} \rho_\kappa^{|\alpha|} |\alpha|!,
\end{aligned}$$

wobei $\gamma_j(\lambda_1, \lambda_2) \in T_j$ und $\gamma_k(\lambda_3, \lambda_4) \in T_k$ sei. Daher sind die Voraussetzungen für Lemma 3.3 erfüllt und es gilt die Abschätzung

$$\begin{aligned}
&\|\kappa(\gamma_j(\cdot), \gamma_k(\cdot)) - \mathcal{I}_p^4 \kappa(\gamma_j(\cdot), \gamma_k(\cdot))\|_{\infty,[0,1]^4} \\
&\qquad \leq C_{\kappa,j,k} 8e(1 + 2\rho_\kappa)\Lambda_p^4(p+1)\left(1 + \frac{1}{\rho_\kappa}\right)^{-(p+1)} \\
&\qquad = C_{\zeta_Q,j,k}\Lambda_p^4(p+1)\left(1 + c_2\zeta_Q\right)^{-(p+1)}.
\end{aligned}$$

Damit ist der Beweis abgeschlossen. □

Da sich wie gezeigt die Kernfunktion besonders gut durch Polynome interpolieren lässt, wollen wir dieses Wissen auf die Quadratur übertragen und werden die vier Integrale im Folgenden durch die Gauss-Quadratur approximieren.

Satz 3.7 *Seien T_j, T_k zwei ζ_Q-zulässige Rechtecke mit zugehörigen Parametrisierungen γ_j, γ_k. Sei weiterhin $\kappa : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}, (\boldsymbol{x}, \boldsymbol{y}) \mapsto \kappa(\boldsymbol{x}, \boldsymbol{y})$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2, s. Sei*

$$\tilde{C}_{\zeta_Q,j,k} := 8e\frac{c_1|T_j||T_k|}{(c_2 \operatorname{dist}(T_j, T_k))^s}\left(1 + \frac{2}{c_2\zeta_Q}\right) = |T_j||T_k|C_{\zeta_Q,j,k}. \tag{3.10}$$

Dann gilt mit $\boldsymbol{\lambda}_j = (\lambda_1, \lambda_2)$ und $\boldsymbol{\lambda}_k = (\lambda_3, \lambda_4)$ für das Integral

$$A_{jk} = \int_{T_j}\int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \tag{3.11}$$

$$= |T_j||T_k| \int_{[0,1]^2}\int_{[0,1]^2} \kappa(\gamma_j(\boldsymbol{\lambda}_j), \gamma_k(\boldsymbol{\lambda}_k)) d\boldsymbol{\lambda}_k d\boldsymbol{\lambda}_j \tag{3.12}$$

und für den durch die Gauss-Quadratur von A_{jk} zum Grad p entstehenden Term

$$(A_p)_{jk} = |T_j||T_k| \sum_{a=0}^{p} w_a \sum_{b=0}^{p} w_b \sum_{c=0}^{p} w_c \sum_{d=0}^{p} w_d \kappa(\gamma_j(\lambda_a, \lambda_b), \gamma_k(\lambda_c, \lambda_d))$$

die Abschätzung

$$|A_{jk} - (A_p)_{jk}| \leq \tilde{C}_{\zeta_Q,j,k}\Lambda_{2p+1}^4 2(p+1)\left(1 + c_2\zeta_Q\right)^{-2(p+1)}. \tag{3.13}$$

Weiterhin gilt für die Konstante $\tilde{C}_{\zeta_Q,j,k}$

$$\tilde{C}_{\zeta_Q,j,k} \leq 8e\frac{c_1}{(c_2\zeta_Q)^s}\max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{2-s}\left(1 + \frac{2}{c_2\zeta_Q}\right)\sqrt{|T_j||T_k|},$$

das heißt für $s \leq 2$ bleibt die Konstante $\tilde{C}_{\zeta_Q,j,k}$ gleichmäßig beschränkt.

Beweis. Wir wissen aus [5, Korollar 6.38], dass die Gauss-Quadratur interpolatorisch mit dem Exaktheitsgrad $2p+1$ ist, unter Berücksichtigung, der hier verwendeten Indizierung beginnend mit 0 statt 1. Deshalb gilt

$$\begin{aligned}
(A_p)_{jk} &= |T_j||T_k| \sum_{a=0}^{p} w_a \sum_{b=0}^{p} w_b \sum_{c=0}^{p} w_c \sum_{d=0}^{p} w_d \kappa(\gamma_j(\lambda_a, \lambda_b), \gamma_k(\lambda_c, \lambda_d)) \\
&= |T_j||T_k| \int_{[0,1]^2}\int_{[0,1]^2} \mathcal{I}_{2p+1}^4 \kappa(\gamma_j(\boldsymbol{\lambda}_j), \gamma_k(\boldsymbol{\lambda}_k)) d\boldsymbol{\lambda}_k d\boldsymbol{\lambda}_j,
\end{aligned}$$

wobei $\mathcal{I}_{2p+1}$ den Chebyshev-Interpolationsoperator vom Grad $2p+1$ bezeichnet. Wegen der Additivität des Integrals und durch Hineinziehen des Betrags erhält man

$$\begin{aligned}
&|A_{jk}-(A_p)_{jk}| \\
&\quad = |T_j||T_k|\Big| \int_{[0,1]^2}\int_{[0,1]^2} \kappa(\gamma_j(\boldsymbol{\lambda}_j),\gamma_k(\boldsymbol{\lambda}_k))d\boldsymbol{\lambda}_k d\boldsymbol{\lambda}_j \\
&\qquad - \int_{[0,1]^2}\int_{[0,1]^2} \mathcal{I}^4_{2p+1}\kappa(\gamma_j(\boldsymbol{\lambda}_j),\gamma_k(\boldsymbol{\lambda}_k))d\boldsymbol{\lambda}_k d\boldsymbol{\lambda}_j\Big| \\
&\quad \leq |T_j||T_k| \int_{[0,1]^2}\int_{[0,1]^2} \left|\kappa(\gamma_j(\boldsymbol{\lambda}_j),\gamma_k(\boldsymbol{\lambda}_k)) - \mathcal{I}^4_{2p+1}\kappa(\gamma_j(\boldsymbol{\lambda}_j),\gamma_k(\boldsymbol{\lambda}_k))\right| d\boldsymbol{\lambda}_k d\boldsymbol{\lambda}_j \\
&\quad \leq |T_j||T_k| \|\kappa(\gamma_j(\cdot),\gamma_k(\cdot)) - \mathcal{I}^4_{2p+1}\kappa(\gamma_j(\cdot),\gamma_k(\cdot))\|_{\infty,[0,1]^4}
\end{aligned}$$

Mithilfe von Satz 3.6 für den Grad $2p+1$ erhalten wir die Behauptung

$$\begin{aligned}
|A_{jk}-(A_p)_{jk}| &\leq |T_j||T_k| C_{\zeta_Q,j,k}\Lambda^4_{2p+1} 2(p+1)\left(1+c_2\zeta_Q\right)^{-2(p+1)} \\
&= \tilde{C}_{\zeta_Q,j,k}\Lambda^4_{2p+1} 2(p+1)\left(1+c_2\zeta_Q\right)^{-2(p+1)}.
\end{aligned}$$

Da die Konstante $\tilde{C}_{\zeta_Q,j,k}$ durch die Netzverfeinerung, aufgrund der Distanz sehr groß werden könnte, untersuchen wir sie an dieser Stelle noch einmal etwas genauer. Es gilt

$$\begin{aligned}
\tilde{C}_{\zeta_Q,j,k} &= 8e\frac{c_1|T_j||T_k|}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(1+\frac{2}{c_2\zeta_Q}\right) \\
&\leq 8e\frac{c_1|T_j||T_k|}{(c_2\zeta_Q\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\})^s}\left(1+\frac{2}{c_2\zeta_Q}\right) \\
&\leq 8e\frac{c_1\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\}^2}{(c_2\zeta_Q\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\})^s}\left(1+\frac{2}{c_2\zeta_Q}\right)\sqrt{|T_j||T_k|} \\
&= 8e\frac{c_1}{(c_2\zeta_Q)^s}\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\}^{2-s}\left(1+\frac{2}{c_2\zeta_Q}\right)\sqrt{|T_j||T_k|}.
\end{aligned}$$

□

Bemerkung 3.8 *Λ_p wächst für Chebyshev-Polynome im wesentlichen logarithmisch in p. Daher konvergiert $(A_p)_{jk}$ für wachsenden Quadraturgrad p exponentiell schnell gegen A_{jk}.*

Im Folgenden wollen wir nun annehmen, dass $T_1, T_2, \ldots, T_n$ Randelemente sind. Dann betrachten wir die Matrizen A, A_p, wobei die Einträge von A genau durch (3.2) gegeben sind und A_p eine A approximierende Matrix ist. Bei der Definition der approximierenden Matrix unterscheiden wir hier zwischen zwei Fällen.

Definition 3.9 *Seien $T_1, T_2, \ldots, T_n$ Rechtecke und $A \in \mathbb{R}^{n\times n}$ gegeben durch* (3.1)*. Dann ist die A approximierende Matrix A_p folgendermaßen definiert.*

- *Sind T_j und T_k ζ_Q-unzulässig, so ist*

  $$(A_p)_{jk} := A_{jk}.$$

- *Sind T_j und T_k ζ_Q-zulässig, so ist mithilfe von Gauss-Quadratur*

  $$(A_p)_{jk} := |T_j||T_k| \sum_{a=0}^{p} w_a \sum_{b=0}^{p} w_b \sum_{c=0}^{p} w_c \sum_{d=0}^{p} w_d \kappa(\gamma_j(\lambda_a,\lambda_b),\gamma_k(\lambda_c,\lambda_d)).$$

Wir wollen nun zeigen, dass die approximierende Matrix bezüglich der Frobenius-Norm für wachsenden Quadraturgrad exponentiell schnell gegen die gegebene Matrix konvergiert. Die Frobenius-Norm ist gegeben durch

$$\|A\|_F := \left(\sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij}^2 \right)^{1/2} \qquad \text{für} \quad A \in \mathbb{R}^{n \times n}.$$

Satz 3.10 *Seien $T_1, T_2, \ldots, T_n$ Rechtecke. Sei $\kappa : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2 und Singularitätsordung $0 \leq s \leq 2$. Sei $\tilde{C}_{\zeta_Q,j,k}$ wie in (3.10) für ζ_Q-zulässsige Rechtecke T_j, T_k und für unzulässige sei $\tilde{C}_{\zeta_Q,j,k} = 0$. Sei $A \in \mathbb{R}^{n \times n}$ eine Matrix, deren Einträge gegeben sind durch*

$$A_{jk} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}$$

und sei A_p die A approximierende Matrix gemäß Definition 3.9. Dann gilt

$$\|A - A_p\|_F \leq 8e \frac{c_1(c_2\zeta_Q + 2)}{(c_2\zeta_Q)^{s+1}} \Lambda_{2p+1}^4 \frac{2(p+1)}{(1 + c_2\zeta_Q)^{2(p+1)}} |\Omega| \max_{\ell=1,\ldots,n} \mathcal{D}(T_\ell)^{2-s}.$$

Beweis. Betrachten wir zunächst die Differenz zwischen A und A_p in einem festen Eintrag $(A - A_p)_{jk}$. Sind T_j und T_k unzulässig, ist die Differenz laut Definition 0 und die Abschätzung mit $\tilde{C}_{\zeta_Q,j,k} = 0$ erfüllt. Sind T_j und T_k hingegen zulässig, können wir Satz 3.7 anwenden. Damit erhalten wir

$$\begin{aligned}
\|A - A_p\|_F^2 &= \sum_{j,k=1}^{n} (A_{jk} - (A_p)_{jk})^2 \\
&\leq \sum_{j,k=1}^{n} \left(\tilde{C}_{\zeta_Q,j,k} \Lambda_{2p+1}^4 2(p+1) \left(1 + c_2\zeta_Q\right)^{-2(p+1)} \right)^2 \\
&\leq \sum_{j,k=1}^{n} \left(8e \frac{c_1 \sqrt{|T_j||T_k|}}{c_2^s \zeta_Q^s \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{s-2}} \left(1 + \frac{2}{c_2\zeta_Q}\right) \Lambda_{2p+1}^4 2(p+1) \left(1 + c_2\zeta_Q\right)^{-2(p+1)} \right)^2 \\
&= \left(8e \frac{c_1}{c_2^s \zeta_Q^s} \left(1 + \frac{2}{c_2\zeta_Q}\right) \Lambda_{2p+1}^4 \frac{2(p+1)}{(1 + c_2\zeta_Q)^{2(p+1)}} \right)^2 \sum_{j,k=1}^{n} \frac{|T_j||T_k|}{\max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{2s-4}} \\
&\leq \left(8e \frac{c_1(c_2\zeta_Q + 2)}{(c_2\zeta_Q)^{s+1}} \Lambda_{2p+1}^4 \frac{2(p+1)}{(1 + c_2\zeta_Q)^{2(p+1)}} \right)^2 |\Omega|^2 \max_{j,k=1,\ldots,n} \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{4-2s}.
\end{aligned}$$

Durch Ziehen der Wurzel auf beiden Seiten, folgt dann die Behauptung. □

3.4 Quadratur über ein Element

Definition 3.11 *Seien $T_j, T_k \in \mathbb{R}^3$ achsenorientierte Rechtecke und sei $\zeta_E > 0$ fest. Dann heißt das Paar (T_j, T_k) ζ_E-zulässig genau dann, wenn*

$$\operatorname{dist}(T_j, T_k) \geq \zeta_E \min\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}. \tag{3.14}$$

Andernfalls heißt das Paar (T_j, T_k) ζ_E-unzulässig.

Satz 3.12 *Seien $T_j, T_k \subseteq \mathbb{R}^3$ zwei ζ_E-zulässige Rechtecke mit $\mathcal{D}(T_j) \leq \mathcal{D}(T_k)$. Sei $\kappa(\cdot,\cdot)$ eine asymptotisch glatte Kernfunktion mit Konstanten $c_1, c_2 > 0$ und Singularitätsordung $s \geq 0$. Dann gilt mit*

$$C_{\zeta_E,j,k} := 8e\frac{c_1}{(c_2 \operatorname{dist}(T_j,T_k))^s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right)$$

die Abschätzung

$$\max_{\boldsymbol{y}\in T_k}\|\kappa(\gamma_j(\cdot),\boldsymbol{y}) - \mathcal{I}_p^2\kappa(\gamma_j(\cdot),\boldsymbol{y})\|_{\infty,[0,1]^2} \leq C_{\zeta_E,j,k}\Lambda_p^2(p+1)\left(1+\sqrt{2}c_2\zeta_E\right)^{-(p+1)}.$$

Beweis. Zunächst definieren wir die Konstanten

$$C_{\kappa,j,k} = \frac{c_1}{(c_2\operatorname{dist}(T_j,T_k))^s} \quad \text{und} \quad \rho_\kappa = \frac{1}{c_2\zeta_E}$$

und können dann die Konstante $C_{\zeta_E,j,k}$ kurz

$$C_{\zeta_E,j,k} = 8eC_{\kappa,j,k}(1+\sqrt{2}\rho_\kappa)$$

schreiben.
Sei $\boldsymbol{y} \in T_k$ fest gewählt. Sei weiterhin γ_j die Parametrisierung zu T_j aus Definition 2.1. So gilt mithilfe von Lemma 3.4

$$\begin{aligned}|\partial_\lambda^\alpha\kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y})| &= \mathcal{D}_{\boldsymbol{a}}(T_j)^{\alpha_1}\mathcal{D}_{\boldsymbol{b}}(T_j)^{\alpha_2}\left|\partial^{t_{jk}(\alpha)}\kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y})\right|,\\ &\leq \mathcal{D}(T_j)^{|\alpha|}\left|\partial^{t_{jk}(\alpha)}\kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y})\right|,\end{aligned}$$

mit Multiindex $\alpha = (\alpha_1,\alpha_2,0,0)$, wobei $|\alpha| > 0$ sei. Ferner gilt mit Definition 3.2 und den Konstanten $C_{\kappa,j,k}, \rho_\kappa$ die Abschätzung

$$\begin{aligned}\mathcal{D}(T_j)^{|\alpha|}\left|\partial^{t_{jk}(\alpha)}\kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y})\right| \leq&\mathcal{D}(T_j)^{|\alpha|}c_1(c_2|\gamma_j(\lambda_1,\lambda_2)-\boldsymbol{y}|)^{-(|\alpha|+s)}|\alpha|!\\ \leq&\mathcal{D}(T_j)^{|\alpha|}c_1(c_2\operatorname{dist}(T_j,T_k))^{-(|\alpha|+s)}|\alpha|!\\ =&\frac{c_1}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(\frac{\mathcal{D}(T_j)}{c_2\operatorname{dist}(T_j,T_k)}\right)^{|\alpha|}|\alpha|!\\ \leq&C_{\kappa,j,k}\rho_\kappa^{|\alpha|}|\alpha|!,\end{aligned}$$

wobei $\gamma_j(\lambda_1,\lambda_2) \in T_j$ sei. Daher sind die Voraussetzungen für Lemma 3.3 erfüllt und es gilt schließlich die Abschätzung

$$\begin{aligned}\|\kappa(\gamma_j(\cdot),\boldsymbol{y}) - \mathcal{I}_p^2\kappa(\gamma_j(\cdot),\boldsymbol{y})\|_{\infty,[0,1]^2} &\leq C_{\kappa,j,k}8e(1+\sqrt{2}\rho_\kappa)\Lambda_p^2(p+1)\left(1+\frac{\sqrt{2}}{\rho_\kappa}\right)^{-(p+1)}\\ &= C_{\zeta_E,j,k}\Lambda_p^2(p+1)\left(1+\sqrt{2}c_2\zeta_E\right)^{-(p+1)}.\end{aligned}$$

□

Satz 3.13 *Seien T_j, T_k zwei ζ_E-zulässige Rechtecke, wobei $\mathcal{D}(T_j) \leq \mathcal{D}(T_k)$ sei , mit zugehörigen Parametrisierungen γ_j, γ_k. Sei weiterhin $\kappa : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R} : (\boldsymbol{x},\boldsymbol{y}) \mapsto \kappa(\boldsymbol{x},\boldsymbol{y})$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2, s. Sei*

$$\tilde{C}_{\zeta_E,j,k} := 8e\frac{c_1|T_j||T_k|}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right) = |T_j||T_k|C_{\zeta_E,j,k}. \tag{3.15}$$

Dann gilt für das Integral

$$A_{jk} = \int_{T_j}\int_{T_k} \kappa(\boldsymbol{x},\boldsymbol{y})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \tag{3.16}$$

$$= |T_j| \int_0^1\int_0^1\int_{T_k} \kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y})d\boldsymbol{y}d\lambda_2 d\lambda_1 \tag{3.17}$$

und für den durch die Gauss-Quadratur von A_{jk} zum Grad p entstehenden Term

$$(A_p)_{jk} = |T_j| \sum_{a=0}^{p} w_a \sum_{b=0}^{p} w_b \int_{T_k} \kappa(\gamma_j(\lambda_a,\lambda_b),\boldsymbol{y}))d\boldsymbol{y}$$

die Abschätzung

$$|A_{jk} - (A_p)_{jk}| \leq \tilde{C}_{\zeta_E,j,k}\Lambda_{2p+1}^2 2(p+1)\left(1+\sqrt{2}c_2\zeta_E\right)^{-2(p+1)}. \tag{3.18}$$

Weiterhin gilt für die Konstante $\tilde{C}_{\zeta_E,j,k}$

$$\tilde{C}_{\zeta_E,j,k} \leq 8e\frac{c_1}{(c_2\zeta_E)^s}\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\}^{2-s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right)\sqrt{|T_j||T_k|}, \tag{3.19}$$

das heißt für $s \leq 1$ bleibt die Konstante $\tilde{C}_{\zeta_E,j,k}$ gleichmäßig beschränkt.

Beweis. Wir wissen aus [5, Korollar 6.38], dass die Gauss-Quadratur interpolatorisch mit dem Exaktheitsgrad $2p+1$ ist, unter Berücksichtigung, der hier verwendeten Indizierung beginnend mit 0 statt 1. Deshalb gilt mit $\boldsymbol{\lambda} = (\lambda_1,\lambda_2)$

$$\begin{aligned}
(A_p)_{jk} &= |T_j| \sum_{a=0}^{p} w_a \sum_{b=0}^{p} w_b \int_{T_k} \kappa(\gamma_j(\lambda_a,\lambda_b),\boldsymbol{y}))d\boldsymbol{y} \\
&= |T_j| \int_0^1\int_0^1\int_{T_k} \mathcal{I}_{2p+1}\kappa(\gamma_j(\lambda_1,\lambda_2),\boldsymbol{y}))d\boldsymbol{y}d\lambda_2 d\lambda_1 \\
&= |T_j| \int_{[0,1]^2}\int_{T_k} \mathcal{I}_{2p+1}\kappa(\gamma_j(\boldsymbol{\lambda}),\boldsymbol{y}))d\boldsymbol{y}d\boldsymbol{\lambda},
\end{aligned}$$

wobei $\mathcal{I}_{2p+1}$ den Chebyshev-Interpolationsoperator vom Grad $2p+1$ bezeichnet. Wegen der Additivität des Integrals und durch Hineinziehen des Betrags erhält man

$$\begin{aligned}
&|A_{jk} - (A_p)_{jk}| \\
&= |T_j| \Big| \int_{[0,1]^2}\int_{T_k} \kappa(\gamma_j(\boldsymbol{\lambda}),\boldsymbol{y})d\boldsymbol{y}d\boldsymbol{\lambda}_j - \int_{[0,1]^2}\int_{T_k} \mathcal{I}_{2p+1}\kappa(\gamma_j(\boldsymbol{\lambda}),\boldsymbol{y})d\boldsymbol{y}d\boldsymbol{\lambda}\Big| \\
&\leq |T_j| \int_{[0,1]^2}\int_{T_k} \Big|\kappa(\gamma_j(\boldsymbol{\lambda}),\boldsymbol{y}) - \mathcal{I}_{2p+1}\kappa(\gamma_j(\boldsymbol{\lambda}),\boldsymbol{y})\Big| d\boldsymbol{y}d\boldsymbol{\lambda} \\
&\leq |T_j||T_k| \sup_{\boldsymbol{y}\in T_k} \|\kappa(\gamma_j(\cdot,\cdot),\boldsymbol{y}) - \mathcal{I}_{2p+1}\kappa(\gamma_j(\cdot,\cdot),\boldsymbol{y})\|_{\infty,[0,1]^2}
\end{aligned}$$

Mithilfe von Satz 3.12 für den Grad $2p+1$ erhalten wir die Behauptung

$$\begin{aligned}
|A_{jk} - (A_p)_{jk}| &\leq |T_j||T_k| C_{\zeta_E,j,k}\Lambda_{2p+1}^2 2(p+1)\left(1+\sqrt{2}c_2\zeta_E\right)^{-2(p+1)} \\
&= \tilde{C}_{\zeta_E,j,k}\Lambda_{2p+1}^2 2(p+1)\left(1+\sqrt{2}c_2\zeta_E\right)^{-2(p+1)}.
\end{aligned}$$

Da die Konstante $\tilde{C}_{\zeta_E,j,k}$ durch die Netzverfeinerung, aufgrund der Distanz sehr groß werden könnte, untersuchen wir sie an dieser Stelle noch einmal etwas genauer. Es gilt

$$\begin{aligned}\tilde{C}_{\zeta_E,j,k} &= 8e\frac{c_1|T_j||T_k|}{(c_2\,\mathrm{dist}(T_j,T_k))^s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right)\\ &\leq 8e\frac{c_1\mathcal{D}(T_j)\mathcal{D}(T_k)}{(c_2\zeta_E\min\{\mathcal{D}(T_j),\mathcal{D}(T_k)\})^s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right)\sqrt{|T_j||T_k|}\\ &\leq 8e\frac{c_1}{(c_2\zeta_E)^s}\max\{\mathcal{D}(T_j),\mathcal{D}(T_k)\}^{2-s}\left(1+\frac{\sqrt{2}}{c_2\zeta_E}\right)\sqrt{|T_j||T_k|}.\end{aligned}$$

□

Lemma 3.14 *Seien T_j, T_k zwei Randstücke mit $\mathrm{dist}(T_j,T_k)>0$, $\kappa(\boldsymbol{x},\boldsymbol{y})$ eine symmetrische asymptotisch glatte Kernfunktion mit*

$$A_{jk}=\int_{T_j}\int_{T_k}\kappa(\boldsymbol{x},\boldsymbol{y})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}}.$$

Dann gilt

$$A_{jk}=A_{kj},$$

insbesondere ist auch die Integrationsreihenfolge aller Integrale beliebig.

Beweis. Aus der Definition 2.4 für den Abstand zweier Rechtecke folgt $T_j\cap T_k=\emptyset$. Da $\kappa(\boldsymbol{x},\boldsymbol{y})$ asymptotisch glatt ist, ist sie auch glatt auf $T_j\times T_k$. Da die Menge beschränkt und abgeschlossen ist, folgt die Kompaktheit aufgrund vom Satz von Heine-Borel. Weiterhin ist κ als stetige Abbildung auf einem kompakten Intervall auf $T_j\times T_k$ integrierbar. Daher gilt der Satz von Fubini

$$\int_{T_j}\int_{T_k}\kappa(\boldsymbol{x},\boldsymbol{y})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}}=\int_{T_k}\int_{T_j}\kappa(\boldsymbol{x},\boldsymbol{y})ds_{\boldsymbol{x}}ds_{\boldsymbol{y}},$$

weshalb die Behauptung folgt. □

Bemerkung 3.15 *Seien T_j, T_k zwei ζ_E zulässige Rechtecke. Ist $\mathcal{D}(T_j)\leq\mathcal{D}(T_k)$, so kann Satz 3.13 angewendet werden um A_{jk} zu approximieren. Im Fall $\mathcal{D}(T_j)>\mathcal{D}(T_k)$ können wir, da für ζ_E-zulässige Elemente auch $\mathrm{dist}(T_j,T_k)>0$ gilt, A_{jk} approximieren, indem wir mithilfe von $A_{jk}=A_{kj}$ und Satz 3.13, A_{kj} berechnen.*

Definition 3.16 *Seien $T_1,T_2,\ldots,T_n$ Rechtecke und $A\in\mathbb{R}^{n\times n}$ gegeben durch* (3.1)*. Dann ist die A approximierende Matrix A_p folgendermaßen definiert.*

- *Sind T_j und T_k ζ_E-unzulässig, so ist*

  $$(A_p)_{jk}:=A_{jk}.$$

- *Sind T_j und T_k ζ_E-zulässig und $\mathcal{D}(T_j)\leq\mathcal{D}(T_k)$, so ist*

  $$(A_p)_{jk}:=|T_j|\sum_{a=0}^{p}w_a\sum_{b=0}^{p}w_b\int_{T_k}\kappa(\gamma_j(\lambda_a,\lambda_b),\boldsymbol{y})d\boldsymbol{y}.$$

- *Sind T_j und T_k ζ_E-zulässig und $\mathcal{D}(T_j) > (T_k)$, so ist*

$$(A_p)_{jk} := |T_k| \sum_{c=0}^{p} w_c \sum_{d=0}^{p} w_d \int_{T_j} \kappa(\boldsymbol{x}, \gamma_k(\lambda_c, \lambda_d)) d\boldsymbol{x}.$$

Satz 3.17 *Seien $T_1, T_2, \ldots, T_n$ Rechtecke. Sei $\kappa : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2 und Singularitätsordung $0 \leq s \leq 1$. Sei $\tilde{C}_{\zeta_E,j,k}$ wie in (3.15) für ζ_E-zulässsige Rechtecke T_j, T_k und für unzulässige sei $\tilde{C}_{\zeta_E,j,k} = 0$. Sei $A \in \mathbb{R}^{n\times n}$ eine Matrix, deren Einträge gegeben sind durch*

$$A_{jk} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}$$

und sei A_p die A approximierende Matrix gemäß Definition 3.16. Dann gilt

$$\|A - A_p\|_F \leq 8e \frac{c_1(c_2\zeta_E + \sqrt{2})}{(c_2\zeta_E)^{s+1}} \Lambda_{2p+1}^2 \frac{2(p+1)}{\left(1 + \sqrt{2}c_2\zeta_E\right)^{2(p+1)}} |\Omega| \max_{\ell=1,\ldots,n} \mathcal{D}(T_\ell)^{2-s}.$$

Beweis. Betrachten wir zunächst die Differenz zwischen A und A_p in einem festen Eintrag $(A - A_p)_{jk}$. Sind T_j und T_k unzulässig, ist die Differenz laut Definition 0 und die Abschätzung mit $\tilde{C}_{\zeta_E,j,k} = 0$ erfüllt. Sind T_j und T_k hingegen zulässig, unterscheiden wir zwei Fälle. Ist $\mathcal{D}(T_j) \leq \mathcal{D}(T_k)$, können wir Satz 3.13 anwenden. Andernfalls ist $A_{jk} = A_{kj}$ durch Lemma 3.14, worauf wir dann Satz 3.13 anwenden können und dadurch dieselbe Abschätzung erhalten. Daraus folgt

$$\begin{aligned}
&\|A - A_p\|_F^2 = \sum_{j,k=1}^{n} (A_{jk} - (A_p)_{jk})^2 \\
&\leq \sum_{j,k=1}^{n} \left(\tilde{C}_{\zeta_E,j,k} \Lambda_{2p+1}^2 2(p+1) \left(1 + \sqrt{2}c_2\zeta_E\right)^{-2(p+1)} \right)^2 \\
&\leq \sum_{j,k=1}^{n} \left(8e \frac{c_1\sqrt{|T_j||T_k|}}{c_2^s \zeta_E^s \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{s-2}} \left(1 + \frac{\sqrt{2}}{c_2\zeta_E}\right) \Lambda_{2p+1}^2 \frac{2(p+1)}{\left(1 + \sqrt{2}c_2\zeta_E\right)^{2(p+1)}} \right)^2 \\
&= \left(8e \frac{c_1}{c_2^s \zeta_E^s} \left(1 + \frac{\sqrt{2}}{c_2\zeta_E}\right) \Lambda_{2p+1}^2 \frac{2(p+1)}{\left(1 + \sqrt{2}c_2\zeta_E\right)^{2(p+1)}} \right)^2 \sum_{j,k=1}^{n} \frac{|T_j||T_k|}{\max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{2s-4}} \\
&\leq \left(8e \frac{c_1(c_2\zeta_E + \sqrt{2})}{(c_2\zeta_E)^{s+1}} \Lambda_{2p+1}^2 \frac{2(p+1)}{\left(1 + \sqrt{2}c_2\zeta_E\right)^{2(p+1)}} \right)^2 |\Omega|^2 \max_{j,k=1,\ldots,n} \max\{\mathcal{D}(T_j), \mathcal{D}(T_k)\}^{4-2s}.
\end{aligned}$$

Durch Ziehen der Wurzel auf beiden Seiten, folgt dann die Behauptung. □

Bemerkung 3.18 *Bei der Randelementemethode mit Galerkin-Verfahren tritt die Steifigkeitsmatrix auf, deren Einträge A_{jk} bis auf einen konstanten Faktor gegeben sind durch*

$$\int_{T_j} \int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{y}}. \tag{3.20}$$

Wir werden anstelle der exakten Matrix die approximative Matrix A_p berechnen, da hierbei Auslöschungseffekte vermieden werden können, welche insbesondere für kleine Randelemente auftreten.

Diese Auslöschungseffekte lassen sich durch ein kleines Beispiel leicht erklären. Hierzu betrachten wir beispielsweise die Stammfunktion des Integrals über ein Element von (3.20)

$$F_{\boldsymbol{x}} = \int \frac{1}{|\boldsymbol{x}-\boldsymbol{y}|} ds_{\boldsymbol{y}}$$

für festes $\boldsymbol{x} \in \mathbb{R}^3$*, inklusive einer geeigneten Parametrisierung* $\gamma_k : [0,1]^2 \to T_k$

$$\tilde{F}_{\boldsymbol{x}} = \int\int \frac{1}{|\boldsymbol{x}-\gamma_k(\boldsymbol{\lambda})|} d\lambda_2 d\lambda_1.$$

Die Stammfunktion ist auch mit der Parametrisierung stetig. Daher gilt für T_k *mit geringem Durchmesser* $\mathcal{D}(T_k)$ *aber* $\tilde{F}_{\boldsymbol{x}}(\lambda) \approx \tilde{F}_{\boldsymbol{x}}(\tilde{\lambda})$ *für alle* $\lambda, \tilde{\lambda} \in [0,1]^2$*, wodurch für den Ausdruck*

$$\tilde{F}_{\boldsymbol{x}}(0,0) - \tilde{F}_{\boldsymbol{x}}(1,0) - \tilde{F}_{\boldsymbol{x}}(0,1) + \tilde{F}_{\boldsymbol{x}}(1,1) = \int_{[0,1]^2} \frac{1}{|\boldsymbol{x}-\gamma_k(\boldsymbol{\lambda})|} d\boldsymbol{\lambda} = \int_{T_k} \frac{1}{|\boldsymbol{x}-\boldsymbol{y}|} ds_{\boldsymbol{y}}$$

starke Auslöschungseffekte auftreten.
Deshalb werden wir für zulässige Randelemente Integrale geeignet durch Gauss-Quadratur ersetzen. Dabei wird der Integrand mehrfach ausgewertet und mit stets positiven Quadraturgewichten multipliziert. Da der Integrand in der Praxis immer das gleiche Vorzeichen hat, werden während der Gauss-Quadratur lediglich Werte gleichen Vorzeichens addiert, wodurch keine Auslöschung auftritt. Deshalb lässt sich beobachten, dass die Auswertung mittels Gauss-Quadratur für zulässige Randstücke genauer ist, als die Berechnung mithilfe der Stammfunktion.
Da wir die Integrationsreihenfolge laut Satz 3.14 beliebig vertauschen dürfen, ist es sinnvoll, über die großen Integrationsbereiche zuerst zu integrieren und die Integration über die kleinen nach außen zu stellen. Dadurch wird jene Operation, durch die starke Auslöschungseffekte auftreten, ans Ende der Berechnung gestellt.
Weiterhin kann die Berechnung durch Quadratur abhängig vom gewählten Quadraturgrad sehr aufwändig werden, welches auf die hohe Anzahl der Auswertungsstellen zurückzuführen ist. Für die Quadratur über beispielsweise zwei Integrale mit einem Quadraturgrad von 8 werden dann schon $8^2 = 64$ *Auswertungen benötigt, über vier Integrale hingegen schon* $8^4 = 4096$*. Deshalb werden wir nicht nur die Strategie betrachten, in der für alle zulässigen Randelemente alle Integrale durch Quadratur ersetzt werden, sondern auch eine, in der nur ein Teil der auftretenden Integrale geeignet durch Gauss-Quadratur ersetzt wird.*

4 Analytische Berechnung

In diesem Abschnitt wollen wir uns mit der analytischen Berechnung des Integrals

$$A_{jk} = \int_{T_j} \int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \in \mathbb{R}^3. \tag{4.1}$$

auf zwei beschränkten, achsenorientierten Rechtecken $T_j, T_k \subseteq \mathbb{R}^3$ beschäftigen. Die im Folgenden auftretenden Stammfunktionen $\int f(x)dx$ werden wir der Einfachheit halber jeweils mit additiver Verschiebung 0 schreiben. Dazu wollen wir [4] folgend zwei Stammfunktionen zitieren, welche durch das Aufspalten des Integrals (4.1) auftreten werden.

Lemma 4.1 *Für die Stammfunktion*

$$g(p; y; x; \lambda) := \int \{(x-y)^2 + \lambda^2\}^p dy$$

mit $\lambda = 0$ gilt die Formel mit beliebigen Skalaren $x, y, p \in \mathbb{R}$

$$(2p+1)g(p; y; x; 0) = \begin{cases} \operatorname{sgn}(y-x)\log|y-x|^{2p} & p = -1/2 \\ (y-x)|y-x|^{2p} & \textit{sonst} \end{cases}$$

falls x für $p \leq -1/2$ außerhalb des betrachteten Integrationsbereichs liegt.
Für beliebige $x, p, y, \lambda \in \mathbb{R}$ mit $\lambda \neq 0$ gilt die Rekursionsformel

$$(2p+1)g(p; y; x; \lambda) = (y-x)\{(y-x)^2 + \lambda^2\}^p + 2p\lambda g(p-1; y; x; \lambda).$$

Im Weiteren benötigen wir noch die Formeln für $p \in \{1/2, 0, -1/2, -1, -3/2\}$

$$\begin{aligned}
g(1/2; y; x; \lambda) &= \frac{y-x}{2}\{(y-x)^2 + \lambda^2\}^{1/2} + \frac{\lambda^2}{2} \operatorname{arsinh} \frac{y-x}{|\lambda|}, \\
g(0; y; x; \lambda) &= y - x, \\
g(-1/2; y; x; \lambda) &= \operatorname{arsinh} \frac{y-x}{|\lambda|}, \\
g(-1; y; x; \lambda) &= \frac{1}{|\lambda|} \arctan \frac{y-x}{|\lambda|}, \\
g(-3/2; y; x; \lambda) &= \frac{y-x}{\lambda^2}\{(y-x)^2 + \lambda^2\}^{-1/2},
\end{aligned}$$

welche alle in [4, Seite 2-3] bewiesen wurden.

Da die Stammfunktion für $p = -1$ in [4, Seite 3] falsch ist verwenden wir die hier hergeleitete. Mithilfe von Substitution durch $z = \frac{y-x}{|\lambda|}$ und $dz = \frac{1}{|\lambda|}dy$ gilt

$$\begin{aligned}
\int \frac{1}{(x-y)^2 + \lambda^2} dy &= \int \frac{1}{\lambda^2((\frac{y-x}{|\lambda|})^2 + 1)} dy = \frac{1}{|\lambda|} \int \frac{1}{z^2+1} dz \\
&= \frac{1}{|\lambda|} \arctan(z) = \frac{1}{|\lambda|} \arctan(\frac{y-x}{|\lambda|}).
\end{aligned}$$

Lemma 4.2 *Des Weiteren werden wir die Stammfunktion*

$$G(p; y_1; y_2; x_1; x_2; \lambda) := \int \int \{(x_1-y_1)^2 + (x_2-y_2)^2 + \lambda^2\}^p dy_2 dy_1$$

benötigen, welche sich für den Fall $p = -3/2$ schreiben lässt als

$$G(-3/2; y_1, y_2; x_1, x_2, 0) = -\frac{\sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}}{(y_1 - x_1)(y_2 - x_2)} \quad \textit{für } \lambda = 0,$$

$$G(-3/2; y_1, y_2; x_1, x_2, \lambda) = -\frac{\operatorname{sgn}\{(y_1 - x_1)(y_2 - x_2)\}}{2|\lambda|} \cdot \arccos\left(\frac{-2(y_1 - x_1)^2(y_2 - x_2)^2}{\{(y_1 - x_1)^2 + \lambda^2\}\{(y_2 - x_2)^2 + \lambda^2\}} + 1\right) \quad \textit{für } \lambda \neq 0.$$

Für alle weiteren relevanten Fälle $p = -3/2 + k$ mit $k \in \mathbb{N}$ können wir folgende Rekursionsformel aufstellen

$$\begin{aligned}(2p+2)G(p; y_1, y_2; x_1, x_2, \lambda) = {} & 2p\lambda^2 G(p-1; y_1, y_2; x_1, x_2, \lambda) \\ & + (y_1 - x_1) g(p; y_2; x_2; \sqrt{(y_1 - x_1)^2 + \lambda^2}) \\ & + (y_2 - x_2) g(p; y_1; x_1; \sqrt{(y_2 - x_2)^2 + \lambda^2}),\end{aligned}$$

welche ebenfalls in [4, Seite 5-7] bewiesen wurden.

4.1 Integral über zwei Elemente

Bei der Berechnung von (4.1) werden wir, aufgrund der speziellen Form des Kerns geometrisch zwischen zwei Fällen unterscheiden. Entweder die beiden Elemente liegen geometrisch in parallelen Ebenen oder in orthogonalen Ebenen.
Das heißt, für parallele Elemente T_j, T_k können wir ohne Beschränkung der Allgemeinheit annehmen, dass mit Definition 2.1

$$\begin{aligned}T_j &= \{\boldsymbol{v} + \lambda_1 a \boldsymbol{a} + \lambda_2 b \boldsymbol{b} \mid \lambda_1, \lambda_2 \in [0,1]\} \\ T_k &= \{\tilde{\boldsymbol{v}} + \tilde{\lambda}_1 \tilde{a} \tilde{\boldsymbol{a}} + \tilde{\lambda}_2 \tilde{b} \tilde{\boldsymbol{b}} \mid \tilde{\lambda}_1, \tilde{\lambda}_2 \in [0,1]\}\end{aligned}$$

$\boldsymbol{a} = \tilde{\boldsymbol{a}}$ und $\boldsymbol{b} = \tilde{\boldsymbol{b}}$ gilt. Wir setzen $\boldsymbol{\delta} = \tilde{\boldsymbol{v}} - \boldsymbol{v} \in \mathbb{R}^3$. Dann gilt für das gesuchte Integral [4, Seite 13] folgend

$$\begin{aligned}&\int_{T_j}\int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &= \int_{T_j}\int_{T_k} \left((x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2\right)^{-1/2} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &= \int_{T_j}\int_{T_k} \left(((x_1 + v_1) - (y_1 + \tilde{v}_1))^2 + ((x_2 + v_2) - (y_2 + \tilde{v}_2))^2 + (v_3 - \tilde{v}_3)^2\right)^{-1/2} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &= \int_0^a \int_0^b \int_0^{\tilde{a}} \int_0^{\tilde{b}} \left((x_1 - y_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2 + \delta_3^2\right)^{-1/2} dy_2 dy_1 dx_2 dx_1.\end{aligned}$$

Lemma 4.3 *Für die Stammfunktion auf parallelen Elementen*

$$F_p(x_1, x_2, y_1, y_2, \boldsymbol{\delta}) := \int\int\int\int \left\{(x_1 - y_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2 + \delta_3^2\right\}^{-1/2} dy_2 dy_1 dx_2 dx_1$$

gilt nach [4, Seite 13]

$$
\begin{aligned}
&F_p(x_1, x_2, y_1, y_2, \boldsymbol{\delta}) \\
&\quad =(x_1 - y_1 - \delta_1)(x_2 - y_2 - \delta_2)G(-1/2; x_1, x_2; y_1 + \delta_1, y_2 + \delta_2, \delta_3) \\
&\qquad - (x_1 - y_1 - \delta_1)g(1/2; x_1; y_1 + \delta_1; \{(x_2 - y_2 - \delta_2)^2 + \delta_3^2\}^{1/2}) \\
&\qquad - (x_2 - y_2 - \delta_2)g(1/2; x_2; y_2 + \delta_2; \{(x_1 - y_1 - \delta_1)^2 + \delta_3^2\}^{1/2}) \\
&\qquad + \frac{1}{3}\{(x_1 - y_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2 + \delta_3^2\}^{3/2}.
\end{aligned} \tag{4.2}
$$

Analog können wir orthogonal liegende Elemente mit Definition 2.1 schreiben als

$$
\begin{aligned}
T_j &= \{\boldsymbol{v} + \lambda_1 a\boldsymbol{a} + \lambda_2 b\boldsymbol{b} \mid \lambda_1, \lambda_2 \in [0,1]\} \\
T_k &= \{\tilde{\boldsymbol{v}} + \tilde{\lambda}_2 \tilde{b}\tilde{\boldsymbol{b}} + \tilde{\lambda}_1 \tilde{a}\tilde{\boldsymbol{a}} \mid \tilde{\lambda}_1, \tilde{\lambda}_2 \in [0,1]\}
\end{aligned}
$$

wobei wir hier $\boldsymbol{b} = \tilde{\boldsymbol{a}}$ und $\boldsymbol{a} \perp \tilde{\boldsymbol{b}}$ annehmen. Wir setzen $\boldsymbol{\delta} = \tilde{\boldsymbol{v}} - \boldsymbol{v} \in \mathbb{R}^3$. Dann gilt nach [4]

$$
\begin{aligned}
&\int_{T_j}\int_{T_k} \frac{1}{|\boldsymbol{x} - \boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\
&= \int_{T_j}\int_{T_k} \left((x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2\right)^{-1/2} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\
&= \int_{T_j}\int_{T_k} \left(((x_1 + v_1) - \tilde{v}_1)^2 + ((x_2 + v_2) - (y_2 + \tilde{v}_2))^2 + (v_3 - (y_3 + \tilde{v}_3))^2\right)^{-1/2} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\
&= \int_0^a \int_0^b \int_0^{\tilde{b}} \int_0^{\tilde{a}} \left((x_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2 + (y_3 - \delta_3)^2\right)^{-1/2} dy_3 dy_2 dx_2 dx_1
\end{aligned}
$$

Lemma 4.4 *Für die Stammfunktion auf orthogonalen Elementen*

$$
\begin{aligned}
&F_o(x_1, x_2, y_2, y_3, \boldsymbol{\delta}) := \\
&\quad \int\int\int\int \left\{(x_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2 + (-y_3 - \delta_3)^2\right\}^{-1/2} dy_3 dy_2 dx_2 dx_1
\end{aligned}
$$

gilt nach [4]

$$
\begin{aligned}
&2F_o(x_1, x_2, y_2, y_3, \boldsymbol{\delta}) \\
&\quad = - G(1/2; y_3, x_1; -\delta_3, \delta_1, x_2 - y_2 - \delta_2) \\
&\qquad - (x_1 - \delta_1)(x_2 - y_2 - \delta_2)G(-1/2; x_2, y_3; y_2 + \delta_2, -\delta_3, x_1 - \delta_1) \\
&\qquad + (x_1 - \delta_1)g(1/2); y_3; -\delta_3, \{(x_1 - \delta_1)^2 + (x_2 - y_2 - \delta_2)^2\}^{1/2}) \\
&\qquad - (y_3 - \delta_3)(x_2 - y_2 - \delta_2)G(-1/2; x_1, x_2; \delta_1, y_2 + \delta_2, -y_3 - \delta_3) \\
&\qquad + (y_3 - \delta_3)g(1/2); x_1; \delta_1, \{(x_2 - y_2 - \delta_2)^2 + (y_3 + \delta_3)^2\}^{1/2}).
\end{aligned} \tag{4.3}
$$

4.2 Bestimmtes Integral

Mithilfe der Stammfunktionen F_p und F_o können wir nun das Integral aus (4.1) für orthogonal und parallel liegende achsenorientierte Rechtecke T und $\tilde{T}$ analytisch exakt lösen. Die Lösung

ist für $F_{p/o} \in \{F_p, F_o\}$, geeignete Grenzen $s_1, s_2, \tilde{s}_1, \tilde{s}_2 \in \mathbb{R}^+$ und $\boldsymbol{\delta} \in \mathbb{R}^3$ gegeben durch

$$
\begin{aligned}
&\int_{T_j}\int_{T_k} \frac{1}{|\boldsymbol{x}-\boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\
&\quad = \int_0^{s_1}\int_0^{s_2}\int_0^{\tilde{s}_1}\int_0^{\tilde{s}_2} \frac{\partial}{\partial y_2}\frac{\partial}{\partial y_1}\frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, y_1, y_2, \boldsymbol{\delta}) dy_2 dy_1 dx_2 dx_1 \\
&= \int_0^{s_1}\int_0^{s_2}\int_0^{\tilde{s}_1} \big(\frac{\partial}{\partial y_1}\frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, y_1, \tilde{s}_2, \boldsymbol{\delta}) \\
&\quad - \frac{\partial}{\partial y_1}\frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, y_1, 0, \boldsymbol{\delta}) \big) dy_1 dx_2 dx_1 \\
&= \int_0^{s_1}\int_0^{s_2} \big(\frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) - \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, \tilde{s}_1, 0, \boldsymbol{\delta}) \\
&\quad - \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, 0, \tilde{s}_2, \boldsymbol{\delta}) + \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} F_{p/o}(x_1, x_2, 0, 0, \boldsymbol{\delta}) \big) dx_2 dx_1 \\
&= \int_0^{s_1} \big(\frac{\partial}{\partial x_1} F_{p/o}(x_1, s_2, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) - \frac{\partial}{\partial x_1} F_{p/o}(x_1, s_2, \tilde{s}_1, 0, \boldsymbol{\delta}) \\
&\quad - \frac{\partial}{\partial x_1} F_{p/o}(x_1, s_2, 0, \tilde{s}_2, \boldsymbol{\delta}) + \frac{\partial}{\partial x_1} F_{p/o}(x_1, s_2, 0, 0, \boldsymbol{\delta}) \\
&\quad - \frac{\partial}{\partial x_1} F_{p/o}(x_1, 0, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) + \frac{\partial}{\partial x_1} F_{p/o}(x_1, 0, \tilde{s}_1, 0, \boldsymbol{\delta}) \\
&\quad + \frac{\partial}{\partial x_1} F_{p/o}(x_1, 0, 0, \tilde{s}_2, \boldsymbol{\delta}) - \frac{\partial}{\partial x_1} F_{p/o}(x_1, 0, 0, 0, \boldsymbol{\delta}) \big) dx_1 \\
&= F_{p/o}(s_1, s_2, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) - F_{p/o}(s_1, s_2, \tilde{s}_1, 0, \boldsymbol{\delta}) - F_{p/o}(s_1, s_2, 0, \tilde{s}_2, \boldsymbol{\delta}) + F_{p/o}(s_1, s_2, 0, 0, \boldsymbol{\delta}) \\
&\quad - F_{p/o}(s_1, 0, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) + F_{p/o}(s_1, 0, \tilde{s}_1, 0, \boldsymbol{\delta}) + F_{p/o}(s_1, 0, 0, \tilde{s}_2, \boldsymbol{\delta}) - F_{p/o}(s_1, 0, 0, 0, \boldsymbol{\delta}) \\
&\quad - F_{p/o}(0, s_2, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) - F_{p/o}(0, s_2, \tilde{s}_1, 0, \boldsymbol{\delta}) - F_{p/o}(0, s_2, 0, \tilde{s}_2, \boldsymbol{\delta}) + F_{p/o}(0, s_2, 0, 0, \boldsymbol{\delta}) \\
&\quad - F_{p/o}(0, 0, \tilde{s}_1, \tilde{s}_2, \boldsymbol{\delta}) + F_{p/o}(0, 0, \tilde{s}_1, 0, \boldsymbol{\delta}) + F_{p/o}(0, 0, 0, \tilde{s}_2, \boldsymbol{\delta}) - F_{p/o}(0, 0, 0, 0, \boldsymbol{\delta})
\end{aligned}
$$

5 Fastreguläre Partionierung in MATLAB

In diesem Abschnitt werden wir die Implementierung etwas genauer betrachten. Hierzu werden wir kurz festhalten wie die Diskretisierung des Netzes aus Kapitel 2.2 in MATLAB und C++umgesetzt wurde. Weiterhin gehen wir auch auf die Implementierung des Verfeinern-Algorithmus aus Kapitel 2.3 ein. Dabei werden wir wichtige Schritte anhand des Codes und kleinen Beispielen hervorheben.

5.1 Datenstruktur

Für die Implementierung in MATLAB und C++ werden wir eine einheitliche Datenstruktur einführen. Die für die Partition $\mathcal{T}_\ell = \{T_1 \dots T_N\}$ benötigten Knoten $\mathcal{K}_\ell = \{\boldsymbol{k}_1 \dots \boldsymbol{k}_M\}$ stellen wir in einer $M \times 3$ Matrix COO dar. Dabei enthält die j-te Zeile die Koordinaten des Knoten $\boldsymbol{k}_j$, d.h. :

$$COO[j, 1:3] = \boldsymbol{k}_j := (x_j, y_j, z_j)^T \text{ wobei } x_j, y_j, z_j \in \mathbb{R}. \tag{5.1}$$

Die Elemente $\mathcal{T}_\ell$ werden wir ebenfalls zeilenweise in einer $N \times 4$ Matrix ELE abspeichern. Dabei soll die i-te Zeile den Indizes der Knoten $\{\boldsymbol{k}_j, \boldsymbol{k}_k, \boldsymbol{k}_\ell, \boldsymbol{k}_m\}$ des Elements T_i entsprechen, also:

$$ELE[i, 1:4] = T_i := (j, k, \ell, m). \tag{5.2}$$

Die Knoten ordnen wir gegen den Uhrzeigersinn an und der Knoten $\boldsymbol{k}_j$ sei der Punkt $\boldsymbol{v}$ aus Definition 2.1.
Für die bessere Handhabung der Elemente beim Verfeinern der Partition, müssen wir auch die Nachbarschaftsrelationen geeignet abspeichern. Im Folgenden werden wir maximal zwei Nachbarn pro Kante eines Elements zulassen, damit das Verhältnis der Elementgrößen benachbarter Elemente beschränkt bleibt. Wir legen also eine $M \times 8$ Matrix für die Indizes der Nachbarelemente an, wobei die i-te Zeile die Nachbarelemente $\{T_{n_1}, \dots, T_{n_8}\}$ zum Element T_i enthält.

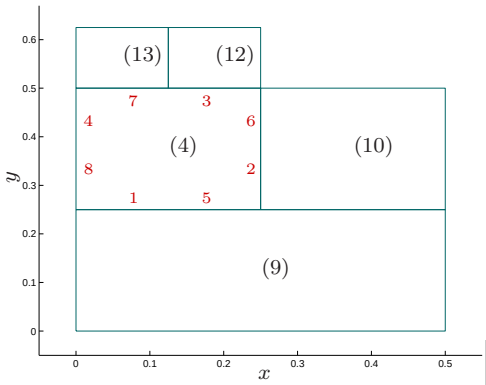
$$NEI[i, 1:8] = N_i := (n_1, \dots, n_8) \tag{5.3}$$

Offensichtlich ist $i \notin N_i$. Wir wollen uns aber noch genauer eine geeignete Anordnung für die Nachbarelemente überlegen. Hierbei bezeichnen wir die Seite $[j, k]$ eines Elements als Seite 1 und gegen den Uhrzeigersinn alle weiteren $[k, \ell], [\ell, m]$ und $[m, j]$ mit 2,3,4. Für den einfacheren Zugriff auf die Elemente einer Seite $s \in \{1, 2, 3, 4\}$ seien die Nachbarelemente zur Seite s unter den Indizes n_s und n_{s+4} abgespeichert. Die Nachbarelemente $\{T_{n_s}, T_{n_{s+4}}\}$ liegen also an der Seite s des Elements. Für Seiten die nur einen Nachbarn T_{n_s} besitzen, setzen wir $n_{s+4} = 0$ und für Seiten mit keinem Nachbarn setzen wir $n_s = n_{s+4} = 0$. Daraus folgt unmittelbar, dass für $n_s = 0$ auch $n_{s+4} = 0$ gilt, die Seite s also keine Nachbarelemente besitzt und umgekehrt folgt aus $n_{s+4} \neq 0$ auch $n_s \neq 0$, womit die Seite s genau zwei Nachbarelemente hat. (Siehe Abbildung 4)

5.2 Verfeinern

Diese Funktion sei die Implementierung von Algorithmus 2.8. Da wir im weiteren Verlauf sowohl adaptive also auch anisotrope Netzverfeinerung zulassen wollen, ist es sinnvoll eine Verfeinerungsfunktion zu implementieren, die alle möglichen Teilungsprozesse auf einem Element realisiert. Dabei sind vier nur wirklich relevant:

1. keine Teilung
2. volle Teilung in vier gleich große Elemente



(a) Lage

Index	n1	n2	n3	n4	n5	n6	n7	n8
4	9	10	13	0	0	0	12	0

(b) Nachbarn : vierte Zeile der NEI Matrix

Abbildung 4: An dieser Stelle wollen wir die Nachbarschaftsrelationen des Elements 4 hervorheben, denn das Element hat an den ersten beiden Seiten jeweils einen Nachbarn, an der dritten Seite zwei Nachbarn und an der vierten Seite keinen. Der Index der Nachbarelemente ist jeweils in den Nachbarschaftsrelationen gespeichert. Wobei bei Seiten mit nur einem Nachbar der Nachbar immer im kleineren Index gespeichert wird. Dass das Element 4 selbst Nachbar einer Seite mit zwei Elementen ist, geht aus den Informationen für Element 4 nicht hervor, ist aber dafür in Element 9 vermerkt. Sollte beispielsweise das Element 12 oder 13 horizontal geteilt werden, so muss klarerweise auch Element 4 mindestens horizontal geteilt werden, welches auch eine horizontale Teilung von Element 9 erzwingen würde.

3. halbe Teilung in zwei gleich große vertikal getrennte Elemente

4. halbe Teilung in zwei gleich große horizontal getrennte Elemente

Zusätzlich wurde auch Typ 5 belegt, welcher als Ergebnis eine volle Teilung vom Typ 2 ausführt, diese aber schrittweise durch eine Teilung vom Typ 3 und anschließend durch jeweils eine Typ 4 Teilung. In der Implementierung wird auch jede vierte Teilung vom Typ 2 durch eine Typ 5 Teilung ausgeführt, da sonst kurzzeitig Seiten mit mehr als zwei Nachbarn auftreten könnten.
Damit jedem Element T_i eine Teilungsart zugeordnet werden kann, haben wir einen Markierungsvektor $marked \in \{1,2,3,4,5\}^M$ eingeführt. Dabei entspricht $marked_i$ der Art der Teilung für das Element T_i. Um isotrope und auch uniforme Teilungen zu erleichtern, kann statt dem Vektor $marked$ auch nur ein Skalar übergeben werden $marked \in \{1,2\dots5\}$, wodurch jedes Element mit der gewählten Art verfeinert wird.
Relevant zum Verfeinern eines Netzes sind also die Koordinaten COO, Elemente ELE sowie die Nachbarschaftsrelationen NEI und der Markierungsvektor $marked$.
Da wir später den Fehlerschätzer berechnen wollen, ist es wichtig, sich zu jedem Element seine Teilelemente zu merken. Dazu legen wir während der Teilung eine $M \times 4$ Matrix $F2S$ an, in der die maximal vier Elementindizes gespeichert werden. Wenn wir also ein Element in vier gleich große Teile verfeinern, so wird das neue Element links unten das erste in der VaterSohn-Matrix sein und alle weiteren folgen gegen den Uhrzeigersinn. Teilen wir ein Element in zwei gleich große Elemente, so werden die doppelt belegten Quadranten auch doppelt eingetragen. Ein gar nicht geteiltes Element wird also vier mal den alten Indizes speichern. Dadurch wird sichergestellt, dass das arithmetische Mittel über die Elemente immer gültig auszuführen ist. In Abbildung 5 stellen wir die VaterSohn-Matrizen der Abbildungen 6 und 7 vor. Durch die Funktion A.2.2

```
[COO_fine, ELE_fine, NEI_fine, F2S] = refineQuad(COO, ELE, NEI, marked);
```

kann die Verfeinerung in MATLAB durchgeführt werden.

Index	e1	e2	e3	e4
1	7	6	5	1
2	8	8	2	2
3	3	3	3	3
4	9	9	4	4

(a) VaterSohn aus 6

Index	e1	e2	e3	e4
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	10	10	4
5	5	5	5	5
6	6	6	6	6
7	13	12	11	7
8	8	8	8	8
9	9	9	9	9

(b) VaterSohn aus 7

Abbildung 5: Anhand der ersten Zeile der VaterSohn-Matrix aus 6 sehen wir anhand der Indizes der neuen Elemente, dass das erste Element in vier Elemente geteilt wurde, also eine isotrope Teilung durchgeführt wurde. Anhand der Indizes in der zweiten Zeile erkennen wir, dass das Element vertikal in zwei neue Elemente geteilt wurde. Das Element 3 wurde hingegen nicht geteilt. Zum Vergleich mit dem Beispiel 7, wurde hier auch die zugehörige VaterSohn-Matrix dargestellt.

5.3 Berechnung der A Matrix

An dieser Stelle wenden wir uns noch einmal der Berechnung der Matrix $A \in \mathbb{R}^{N\times N}$ zu, deren Einträge bestimmt sind durch das Integral

$$A_{jk} = \int_{T_j}\int_{T_k} \frac{1}{4\pi}\frac{1}{|\boldsymbol{x}-\boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}. \tag{5.4}$$

Da die Berechnung aller Einträge sehr aufwändig ist, haben wir die entsprechenden Funktionen nicht direkt in MATLAB implementiert, sondern mithilfe der MEX-Schnittstelle in C++. MATLAB benötigt dazu die Bibliothek „mex.h“ und die spezielle MEX-Funktion als Einstiegspunkt mit der Gestalt

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]); .
```

Dabei steht `nlhs` für die erwartete Anzahl der Rückgabe-Parameter, welche im Array `*plhs[]` abgespeichert werden müssen. Im Array `*prhs[]` werden die Eingabe-Parameter von MATLAB übergeben, dessen Anzahl über `nrhs` abgefragt werden kann. Weiterhin stehen noch wichtige MEX-Funktionen zum Umwandeln der Parameter sowie dem Zugriff auf MATLAB-Funktionen und Ausgaben zur Verfügung.
Die Implementierung des Integrals (5.4) wurde in `mex_build_V` (Anhang A.1.1) mithilfe der beiden Bibliotheken `slpRectangle` (Anhang A.1.3, A.1.2) und `gauss` (Anhang A.1.4) umgesetzt. Innerhalb von MATLAB kann der Code dann kompiliert werden mit

```
mex 'mex_build_V.cpp' 'slpRectangle.cpp' CFLAGS="\$CFLAGS -O2 -fopenmp"...
    CXXFLAGS="\$CXXFLAGS -O2 -fopenmp" LDFLAGS="\$LDFLAGS -O2 -fopenmp".
```

Nach erfolgreichem Kompilieren wurde eine `mex_build_V.mexa64` Datei für ein 64bit System erstellt, welche dann in MATLAB über

```
A = mex_build_V(coo,ele,zeta,typ);
```

aufgerufen werden kann. Hierbei steht, wie in der Datenstruktur beschrieben, `coo` für die Koordinatenmatrix und `ele` für die Elementmatrix. Weiterhin sei `zeta` ein Array mit den Einträgen `zeta` $= (q, \zeta_Q, \zeta_E)$. Mithilfe des Parameters $q \in \{0,1,2,3,4,5\}$ kann die Anzahl 2^q der Auswertungsstellen für die Gauss-Quadratur gewählt werden. Die beiden Parameter $\zeta_Q, \zeta_E \in \mathbb{R}^+$

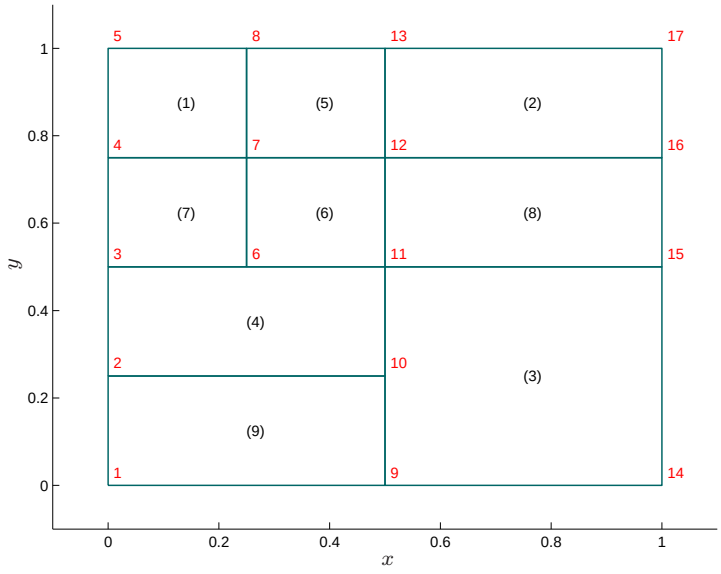
werden in den Zulässigkeitsbedingungen verwendet. Der Parameter `typ` $\in \{1, 2, 3, 4\}$ steht für die Art der Berechnung.
Insgesamt wurden vier Arten der Berechnung implementiert.

1. Bei der voll analytischen Berechnung von A werden
   - alle Elemente analytisch mittels Stammfunktion berechnet.
2. In der vollen Quadratur für zulässige Elemente werden
   - ζ_Q-zulässige Elemente mit Gauss-Quadratur über alle Integrale
   - und alle anderen analytisch mittels Stammfunktion berechnet.
3. Bei der geeigneten Quadratur für zulässige Elemente werden
   - ζ_E-zulässige Elemente mit Gauss-Quadratur über ein Element,
   - ζ_Q-zulässige Elemente mit Gauss-Quadratur über alle Integrale
   - und alle anderen analytisch mittels Stammfunktion berechnet.
4. In der Quadratur über ein Element für zulässige Elemente werden
   - ζ_E-zulässige Elemente mit Gauss-Quadratur über ein Element
   - und alle anderen analytisch mittels Stammfunktion berechnet.

Da der Code sehr umfangreich geworden ist, werden wir an dieser Stelle nur einige Details hervorheben.
Wie schon erwähnt, dient die `mex_build_V.cpp` als Einstiegspunkt für MATLAB. In ihr haben wir die Auswertung der Parameter sowie das Ermitteln der Integrationsgrenzen implementiert. Weiterhin iterieren wir hier mittels `for`-Schleifen auch über die Einträge der Matrix, um dann die geeignete Berechnungsfunktion des Eintrags A_{jk} aufzurufen.
Da die Berechnungen der einzelnen Matrixeinträge unabhängig voneinander sind, haben wir an dieser Stelle die `for`-Schleifen zur Nutzung von paralleler Berechnung auf mehreren Kernen optimiert. Dafür nutzen wir die freie Bibliothek `omp.h`.
Die Implementierung der Berechnung eines Eintrags A_{jk} befindet sich in der `slpRectangle.cpp`. Auch hier haben wir zwischen parallel und orthogonal liegenden Elementen unterschieden. Weiterhin haben wir versucht die Funktionen möglichst direkt von der Analysis aus Kapitel 4 in C++ zu implementieren und auf mögliche Optimierungen zur Fehlervermeidung zu verzichten.

5.4 Beispielnetz mit Verfeinerung

An dieser Stelle wollen wir noch einmal das Netz des Quadrats aus Abbildung 8a betrachten. Wir wenden dazu die Netzverfeinerung aus Kapitel 5.2 mit dem Markierungsvektor $marked = (2, 3, 1, 3)$ auf das Startnetz an und erhalten dadurch das Netz aus Abbildung 6a. Die drei zugehörigen Matrizen COO, ELE und NEI, welche in Kapitel 5.1 genauer beschrieben wurden, finden wir in den Abbildungen 6b,6c,6d.
Wenn wir nun in diesem Netz das Element 7 durch eine Teilung vom Typ 2 weiter verfeinern, muss auch das Element 4 geteilt werden. Abbildung 7 stellt dieses neue Netz dar und zeigt, dass der Algorithmus 2.8 die nötigen Teilungen vornimmt.



(a) Lage

Index	x1	x2	x3
1	0	0	0
2	0	0.25	0
3	0	0.5	0
4	0	0.75	0
5	0	1	0
6	0.25	0.5	0
7	0.25	0.75	0
8	0.25	1	0
9	0.5	0	0
10	0.5	0.25	0
11	0.5	0.5	0
12	0.5	0.75	0
13	0.5	1	0
14	1	0	0
15	1	0.5	0
16	1	0.75	0
17	1	1	0

(b) Koordinatenmatrix

Index	c1	c2	c3	c4
1	4	7	8	5
2	12	16	17	13
3	9	14	15	11
4	2	10	11	3
5	7	12	13	8
6	6	11	12	7
7	3	6	7	4
8	11	15	16	12
9	1	9	10	2

(c) Elementmatrix

Index	n1	n2	n3	n4	n5	n6	n7	n8
1	7	5	0	0	0	0	0	0
2	8	0	0	5	0	0	0	0
3	0	0	8	9	0	0	0	4
4	9	3	7	0	0	0	6	0
5	6	2	0	1	0	0	0	0
6	4	8	5	7	0	0	0	0
7	4	6	1	0	0	0	0	0
8	3	0	2	6	0	0	0	0
9	0	3	4	0	0	0	0	0

(d) Nachbarschaftsrelationen

Abbildung 6: Dieses Beispielnetz entsteht durch die Verfeinerung von Netz 8 mit dem Markierungsvektor $marked = (2, 3, 1, 3)$. Zum Verfeinern wurde der Algorithmus 2.8 mit der Implementierung aus A.2.2 verwendet. In schwarz wurde der Index der Elemente und in rot der Index der Koordinaten eingezeichnet.

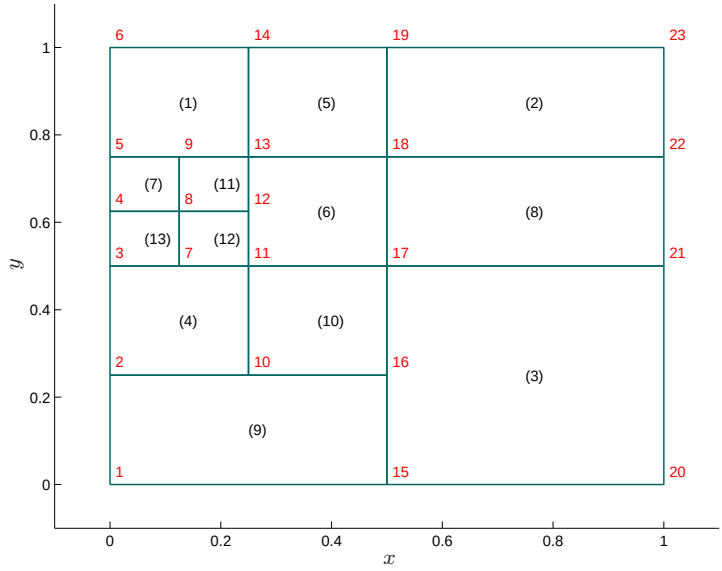


Abbildung 7: Dieses Netz entsteht durch die Verfeinerung von Element 7 aus dem Netz 6. In schwarz wurde der Index der Elemente und in rot der Index der Koordinaten eingezeichnet. Es zeigt, dass zusätzlich Elemente verfeinert werden, um die Forderung von maximal drei Knoten auf einer Kante zu erfüllen.

6 Numerische Experimente

In diesem Abschnitt werden wir anhand von zwei Beispielgeometrien die gezeigten Zusammenhänge genauer untersuchen. Hierzu werden wir zunächst einige Fehlerschätzer definieren, welche auch lokale Aussagen auf den Netzen zulassen. Weiterhin werden wir eine Strategie zum Markieren von Elementen vorstellen, die die Definition eines adaptiven Algorithmus ermöglicht. Da wir durch verschiedene Strategien die Stabilität der Berechnungen sicherstellen wollen, werden wir unsere Berechnungen auf das Lösen der Gleichung

$$V\phi = 1$$

mit konstanter rechter Seite beschränken.
Alle Experimente wurden auf einem Rechner mit acht Intel(R) Xeon(R) Prozessoren mit jeweils 2.5GHz durchgeführt. Auf dem Betriebsystem Ubuntu 3.0.0-32-server mit 32GB verfügbaren Arbeitsspeicher wurde die gcc Version 4.6.1 und MATLAB R2011a (7.12.0) 64bit verwendet. Während den Berechnungen konnte eine Systemauslastung von etwa 700 % beobachtet werden, welches der Benutzung von sieben Kernen entspricht.

6.1 Fehlerschätzer

Mithilfe des Galerkin-Verfahrens berechnen wir nur eine approximative Lösung von (2.2), für die wir eine Aussage über die Genauigkeit der Lösung treffen wollen. Da wir für den Fehler $|||\phi - \phi_\ell|||$ nur die Galerkin-Lösung ϕ_ℓ kennen und die exakte Lösung ϕ unbekannt ist, werden wir im Folgenden verschiedene Fehlerschätzer betrachten. Im Wesentlichen werden wir hierzu die $h-h/2$ Strategie aus [3] verwenden. Im Folgenden bezeichnen wir mit $\hat{\mathcal{T}}_\ell$ das Gitter, welches entsteht, wenn das Gitter $\mathcal{T}_\ell$ uniform, also entlang aller Kanten geteilt wird. Weiterhin bezeichne ϕ die exakte Lösung des Galerkin-Verfahrens und ϕ_ℓ die Galerkin-Lösung zum Gitter $\mathcal{T}_\ell$ sowie $\hat{\phi}_\ell$ die Galerkin-Lösung zum uniformen Gitter $\hat{\mathcal{T}}_\ell$.

Definition 6.1 *Es bezeichne ϕ die exakte Lösung von (2.2), ϕ_ℓ die Galerkin-Lösung von (2.4) auf dem Gitter $\mathcal{T}_\ell$ und $\hat{\phi}_\ell$ die Galerkin-Lösung auf dem uniform verfeinerten Gitter $\hat{\mathcal{T}}_\ell$. Dann gilt, der Schätzer*

$$\eta_\ell := |||\hat{\phi}_\ell - \phi_\ell|||$$

ist effizient

$$\eta_\ell \leq |||\phi - \phi_\ell|||$$

und unter der Saturationsannahme

$$\|\phi - \hat{\phi}_\ell\| \leq C_{sat}\|\phi - \phi_\ell\| \quad \textit{mit } 0 < C_{sat} < 1$$

zuverlässig

$$|||\phi - \phi_\ell||| \leq \frac{1}{\sqrt{1 - C_{sat}^2}}\eta_\ell.$$

Der Schätzer ist berechenbar, liefert aber keine lokalen Beiträge und kann daher nicht unmittelbar verwendet werden, um einen adaptiven Algorithmus zu steuern. Hierzu definieren wir weitere Schätzer. Für jedes $T_j \in \mathcal{T}_\ell$ bezeichnen wir mit $\varrho_j := \min\{\mathcal{D}_{\boldsymbol{a}}(T_j), \mathcal{D}_{\boldsymbol{b}}(T_j)\} > 0$ die Länge der kürzesten Seite von T_j und mit $h_j := \mathcal{D}(T_j) > 0$ den Durchmesser von T_j. Weiterhin definieren wir die lokalen Netzweiten $\varrho, h \in L^\infty$ durch $\varrho_{|T_j} = \varrho_j$ und $h_{|T_j} = h_j$.

Definition 6.2 (A-posteriori Fehlerschätzer) *Es bezeichne ϕ die exakte Lösung von (2.2), ϕ_ℓ die Galerkin-Lösung von (2.4) auf dem Gitter $\mathcal{T}_\ell$ und $\hat{\phi}_\ell$ die Galerkin-Lösung auf dem uniform verfeinerten Gitter $\hat{\mathcal{T}}_\ell$. Dann gilt nach [3, Theorem 3.2 und 3.4], die Fehlerschätzer*

$$\mu_\ell = \|\varrho^{1/2}(\hat{\phi}_\ell - \phi_\ell)\|_{L^2(\Gamma)} \tag{6.1}$$

$$\tilde{\eta}_\ell = |\!|\!|\hat{\phi}_\ell - \Pi_\ell \hat{\phi}_\ell|\!|\!| \tag{6.2}$$

$$\tilde{\mu}_\ell = \|\varrho^{1/2}(\hat{\phi}_\ell - \Pi_\ell \hat{\phi}_\ell)\|_{L^2(\Gamma)}, \tag{6.3}$$

wobei Π_ℓ die L_2 Projektion auf $P^0(\mathcal{T}_\ell)$ ist,

- *sind mit Konstanten $C_3, C_4 > 0$ äquivalent*
  $\eta_\ell \leq \tilde{\eta}_\ell \leq C_4 \|(h/\varrho)^{1/2}\|_{L^\infty(\Gamma)} \tilde{\mu}_\ell$ und $\tilde{\mu}_\ell \leq \mu_\ell \leq \sqrt{2} C_3 \eta_\ell$,
- *sind effizient und*
- *sind unter Saturationsannahme auch zuverlässig.*

6.2 Markieren

Im adaptiven Algorithmus werden wir die Elemente abhängig vom Fehlerschätzer $\tilde{\mu}$ verfeinern. Dazu wählen wir mithilfe der Dörfler-Markierung [2] eine Teilmenge aus.

Definition 6.3 (Dörfler-Markierung) *Bestimme für feste Konstante $\theta \in (0, 1]$ die Menge $M_\ell \subseteq T_\ell$ mit minimaler Kardinalität*

$$\theta \sum_{T \in T_\ell} \tilde{\mu}_\ell(T)^2 \leq \sum_{T \in M_\ell} \tilde{\mu}_\ell(T)^2.$$

Um die Symmetrie des Netzes zu erhalten, bestimme weiterhin für feste Konstante $\vartheta \in (0, 1)$ die kleinste Teilmenge $\tilde{M}_\ell \subseteq T_\ell \backslash M_\ell$ für die gilt

$$\vartheta \sum_{T \in M_\ell} \tilde{\mu}_\ell(T)^2 < \sum_{T \in \tilde{M}_\ell} \tilde{\mu}_\ell(T)^2.$$

Die Menge der markierten Elemente ist dann gegeben durch $M_\ell \cup \tilde{M}_\ell$.

Im Folgenden werden wir $\vartheta = 10^{-2}$ wählen. Da wir im Adaptiven Algorithmus auch anisotrope Verfeinerungen zulassen wollen, definieren wir an dieser Stelle eine Auswahlstrategie zum Bestimmen der Verfeinerungsart für jedes Element.

Definition 6.4 *Seien $\phi_j^{(1)}, \ldots, \phi_j^{(4)}$ die Lösungen der isotropen Verfeinerung $T_j^{(1)}, \ldots, T_j^{(4)} \in \hat{\mathcal{T}}_\ell$ von $T_j \in T_\ell$, dann sei C_j zum Element T_j definiert durch*

$$\begin{pmatrix} C_j^{(1)} \\ C_j^{(2)} \\ C_j^{(3)} \\ C_j^{(4)} \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \phi_j^{(1)} \\ \phi_j^{(2)} \\ \phi_j^{(3)} \\ \phi_j^{(4)} \end{pmatrix}.$$

Dann ist mit fester Konstante $\nu \in [0, 1)$ die Art der Markierung $marked_j$ zu einem Element $T_j \in \tilde{M}_\ell$ wie in [3, Kapitel 4.5] bestimmt durch

$$marked_j := \begin{cases} 3 & (vertikal) \quad \text{falls } \nu |C_j^{(3)}| \geq \sqrt{|C_j^{(2)}|^2 + |C_j^{(4)}|^2} \\ 4 & (horizontal) \quad \text{falls } \nu |C_j^{(4)}| \geq \sqrt{|C_j^{(2)}|^2 + |C_j^{(3)}|^2} \\ 2 & (isotrop) \quad \text{sonst.} \end{cases}$$

Weiterhin sei $marked_j = 1$ für alle $T_j \in \mathcal{T}_\ell \backslash \tilde{M}_\ell$.

Die Funktion A.2.3

```
marked = mark(xF2S, tmu, theta, nu);
```

implementiert die Definitionen zum Bestimmen der Markierung. Hierbei seien die Einträge der Matrix $xF2S \in \mathbb{R}^{N\times 4}$ geben durch $(xF2S)_{ij} = \langle \hat{\phi}_\ell, \chi_{(F2S_{ij})} \rangle$.

6.3 Adaptivität

Mithilfe der oben definierten Funktionen ist es uns nun möglich den Ablauf der Berechnungen zusammen zu fassen.

Algorithmus 6.5 (Adaptives Verfahren) *Sei $\theta, \nu \in (0,1)$ fest gewählt und das Startnetz $\mathcal{T}_\ell$ gegeben.*

(i). Verfeinere $\mathcal{T}_\ell$ um $\hat{\mathcal{T}}_\ell$ zu erhalten

(ii). Berechne die Galerkin-Lösung $\hat{\phi}_\ell \in P^0(\hat{\mathcal{T}}_\ell)$ von (2.4)

(iii). Berechne Fehlerschätzer $\tilde{\mu}_i := \|\varrho^{1/2}(\hat{\phi}_\ell - \Pi_\ell \hat{\phi}_\ell)\|$

(iv). Wähle Teilmenge $M_\ell \subseteq \mathcal{T}_\ell$ durch Definition 6.3 und 6.4

(v). Verfeinere mindestens die markierten Elemente M_ℓ durch Algorithmus 2.8, um $\mathcal{T}_{\ell+1}$ zu erhalten

(vi). $\ell \mapsto \ell + 1$, gehe zu (i)

6.4 Beispiel Quadrat-Schirm

Im Folgenden werden wir die Laplace-Gleichung

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega \subset \mathbb{R}^3, \\ u_{|\Gamma} &= 1 \quad \text{auf } \Gamma, \end{aligned} \tag{6.4}$$

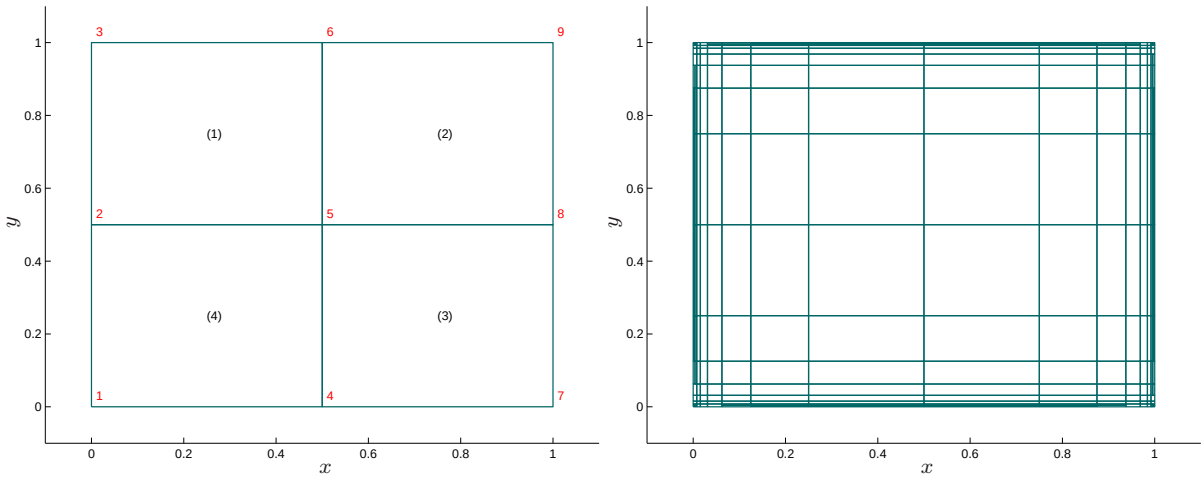
mit dem Startnetz aus Abbildung 8a, einem Quadrat in der $z = 0$ Ebene, genauer betrachten. Die vier Eckpunkte des Quadrat-Schirms sind hierbei gegeben durch

$$\{(0,0,0), (1,0,0), (1,1,0), (0,1,0)\}.$$

Die Lösung ϕ kann nicht exakt bestimmt werden, weist aber an den vier Eckpunkten Singularitäten auf. Wir werden aber trotzdem die Energienorm als Referenzlösung mithilfe des *Aitken'schen* Δ^2-Verfahren bestimmen. Das Verfahren dient zum Beschleunigen der Konvergenz von Folgen. Damit schätzen wir für die exakte Lösung ϕ die Energienorm $|||\phi|||^2 \approx 4.609193$.

6.4.1 Vergleich verschiedener Verfeinerungsstrategien

Zunächst wollen wir drei Verfeinerungs-Strategien genauer untersuchen. Hierzu betrachten wir zum einen die Strategie „uniform“ ($\theta = 1, \nu = 0$), bei der das verfeinerte Netz $\mathcal{T}_{\ell+1}$ durch isotrope Verfeinerung aller Elemente entsteht, also jedes Element wird in vier gleich große Elemente geteilt. In der zweiten Strategie „adaptiv isotrop“ ($\theta = 0.5, \nu = 0$) werden wir zulassen, dass nicht alle Elemente verfeinert werden, also nur eine Teilmenge wird jeweils in vier gleich große Elemente geteilt. Und in der letzten Strategie „adaptiv anisotrop“ ($\theta = 0.5, \nu = 0.5$) werden wir außerdem anisotrope Teilungen zulassen, also ein Teil der Elemente wird geeignet in

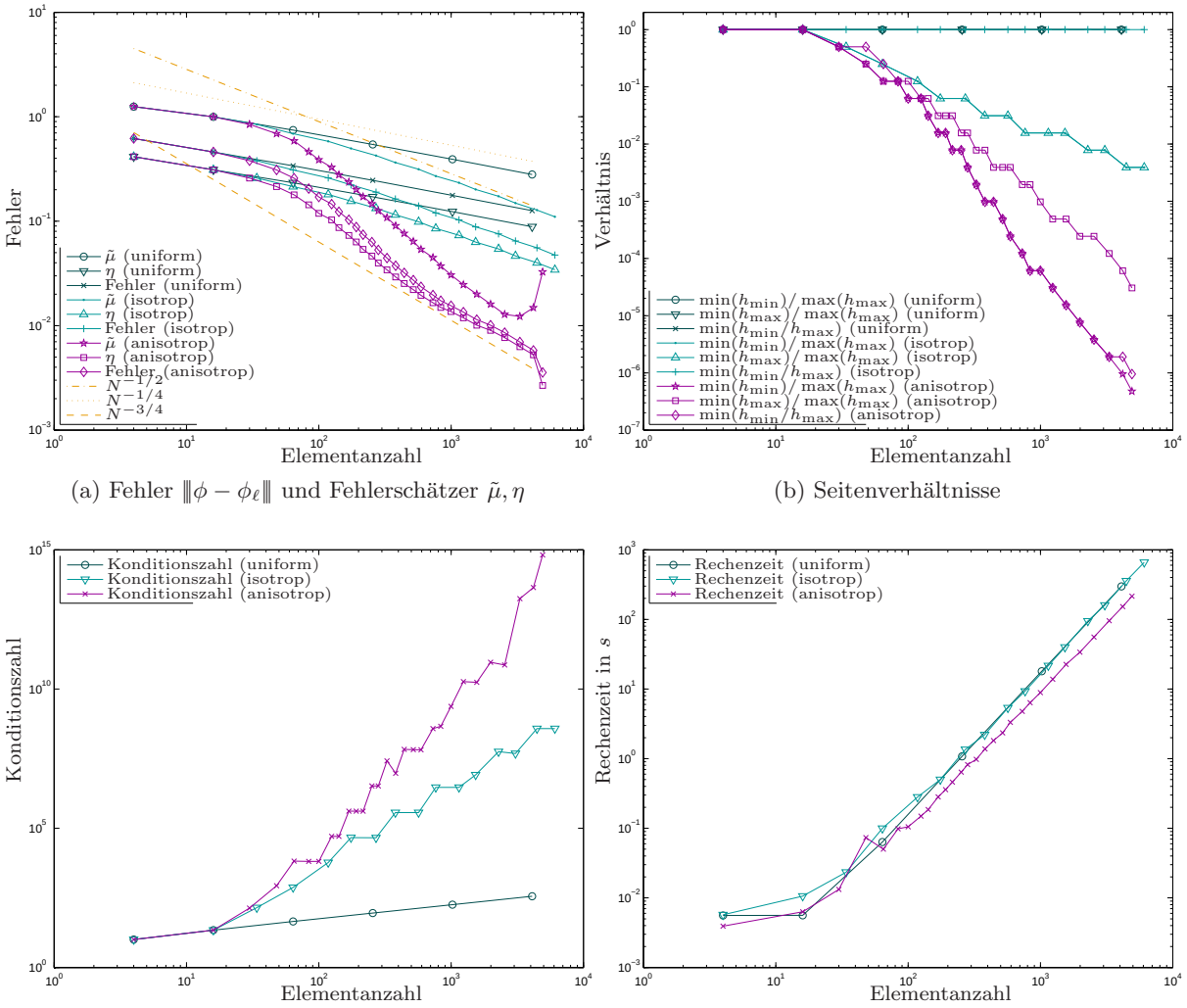


(a) Startnetz

(b) Anisotropes Netz nach 12 Schritten mit 252 Elementen

Abbildung 8: Das Startnetz zeigt die Geometrie Quadrat in der $z = 0$ Ebene für die Beispiele 6.4. Anhand des Netzes der anisotropen Verfeinerung lässt sich gut erkennen, dass die Geometrie wie erwartet vorwiegend an den Singularitäten, also am Rand verfeinert wird.

zwei oder vier gleich große Elemente geteilt. Für alle drei Strategien werden wir den Algorithmus 6.5 mit entsprechenden Parametern θ, ν zum Lösen von (6.4) verwenden.
In Abbildung 9a betrachten wir für die jeweilige Strategie jeweils die Fehlerschätzer $\tilde{\mu}$ und η sowie den Fehler gegenüber der tatsächlichen Lösung. Die Werte wurden jeweils über die Elementanzahl des $\mathcal{T}_\ell$ Netzes aufgetragen. Anhand der dunkelgrünen Linien erkennen wir, dass der Fehler bei der „uniformen“ Strategie mit etwa $\mathcal{O}(N^{-1/4})$ gegen 0 strebt. Gerechnet wurde hier bis zu einer Elementanzahl des $\mathcal{T}_\ell$ Netzes von etwa 3000, für das der Fehler der Energienorm im Bereich von 1 liegt.
Anhand der Linien in zyan, beobachten wir eine schnellere Konvergenz der „adaptiv isotropen“ Strategie. Denn hier lässt sich eine Konvergenzrate von $\mathcal{O}(N^{-1/2})$ gegen 0 erkennen. Bei dieser Strategie erreichen wir für die gleiche Elementanzahl des $\mathcal{T}_\ell$ Gitters schon einen Fehler im Bereich von 0.1.
Betrachten wir nun die Strategie „adaptiv anisotrop“ in magenta, so beobachten wir eine kurzzeitig sehr starke Konvergenz, welche dann gegen eine sehr gute Konvergenzrate von $\mathcal{O}(N^{-3/4})$ strebt. Hierbei erkennen wir, dass der Fehler der Energienorm schon im Bereich von 0.01 für die gleiche Elementanzahl liegt. Jedoch sehen wir auch, dass der Fehlerschätzer $\tilde{\mu}$ ab dieser Elementanzahl seine Konvergenzrate verliert und damit unzuverlässig wird.
Weiterhin können wir für alle drei Strategien anhand der Parallelität der Fehlerschätzer zum tatsächlichen Fehler, die Effizienz und Zuverlässigkeit der Fehlerschätzer erkennen. Ebenso ist die Äquivalenz des $h - h/2$ Schätzers zum lokalen $\tilde{\mu}$ Schätzers, aufgrund der Parallelität zu beobachten. Außerdem beschreiben die Fehlerschätzer den tatsächlichen Fehler auch in der Größenordnung sehr gut.
In Abbildung 9b betrachten wir bestimmte Eigenschaften zwischen den Seitenverhältnissen der Elemente aus dem $\mathcal{T}_\ell$ Netz für die jeweilige Strategie. $h_{\min}$ steht hierbei für die kürzere Seite eines Rechtecks $T \in \mathcal{T}_\ell$ und $h_{\max}$ für die längere der beiden Seiten. Weiterhin verstehen wir das Minimum min() und Maximum max() über alle Elemente $T \in \mathcal{T}_\ell$. Wir werden nun das Verhältnis der kleinsten kurzen Seite gegenüber der größten langen Seite $\min(h_{\min})/\max(h_{\max})$, das Verhältnis der kleinsten langen gegenüber der größten langen Seite $\min(h_{\max})/\max(h_{\max})$



(a) Fehler $\|\phi - \phi_\ell\|$ und Fehlerschätzer $\tilde{\mu}, \eta$

(b) Seitenverhältnisse

(c) Konditionszahl der $\hat{V}_\ell$ Matrix

(d) Berechnungszeit

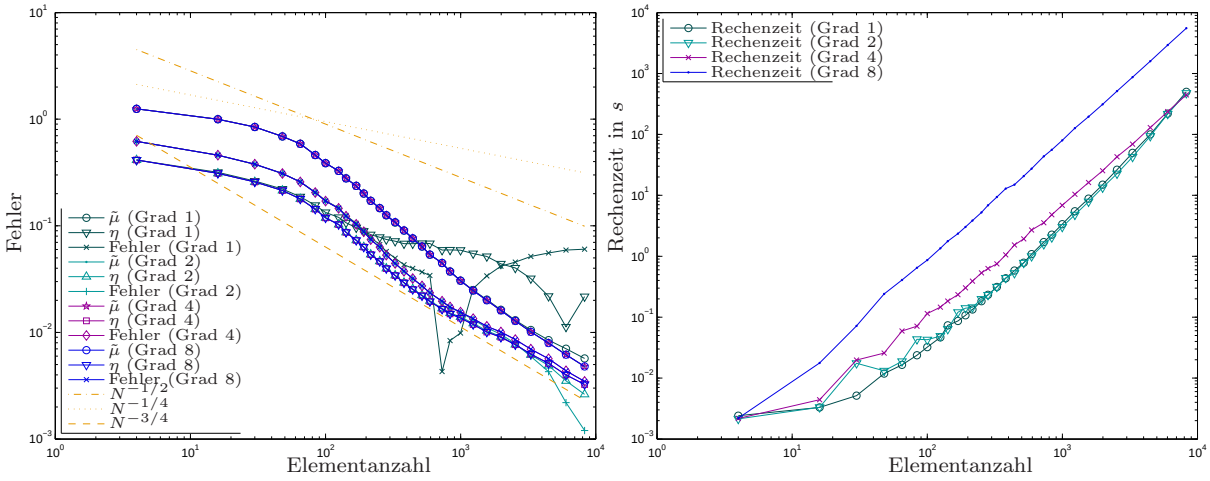
Abbildung 9: Vergleich der Verfeinerungsstrategien auf dem Quadrat in Beispiel 6.4.1

sowie das kleinste Verhältnis der kurzen gegenüber der langen Seiten $\min(h_{\min}/h_{\max})$ für die drei Strategien genauer betrachten.
Bei der „uniformen“ Strategie, dargestellt durch die dunkelgrüne Linien, sind wie erwartet, alle Verhältnisse gleich 1, da alle Elemente deckungsgleich sind.
Bei der „adaptiv isotropen“ Strategie in zyan ist am Verhältnis $\min(h_{\min}/h_{\max}) = 1$ gut zu erkennen, dass alle Elemente Quadrate sind. Die beiden anderen Verhältnisse zeigen, dass bei zunehmender Elementanzahl die Differenz der Elementgrößen zunimmt.
Anhand der magenta farbenen Linien, also der „adaptiv anisotropen“ Strategie, beobachten wir am kleiner werdenden Verhältnis $\min(h_{\min}/h_{\max})$, dass mit zunehmender Elementanzahl lange schmale Elemente entstehen. Am kleiner werden der anderen beiden Verhältnisse erkennen wir auch hier, dass die Differenz der Elementgrößen zunimmt.
Um auch die Stabilität der drei Strategien untersuchen zu können, sehen wir in der Abbildung 9c die Konditionszahlen der Matrix $\hat{V}_\ell$ in Abhängigkeit der Elementanzahl. Für die „uniforme“ Strategie in dunkelgrün erkennen wir sehr gute Konditionszahlen, wissen aber, dass auch der Fehler der Galerkin-Lösung nur langsam gegen 0 konvergiert. Die Konditionszahlen der „adaptiv anisotropen“ Strategie in magenta wachsen hingegen am schnellsten und steigen bei etwa 3000

Elementen sogar sprunghaft an.
Weiterhin können wir in Abbildung 9d die benötigte Rechenzeit für einen Berechnungsschritt ablesen. In einem Berechnungsschritt wird die Matrix $\hat{V}_\ell$ und V_ℓ aufgestellt und die Galerkin-Lösung inklusive aller Fehlerschätzer berechnet. Hierbei fällt auf, dass die Wahl der Strategie keinen Einfluss auf die benötigte Rechenzeit hat, sondern nur die Anzahl der Elemente. Für die Berechnung mit 3000 Elementen benötigen alle drei Strategien etwa 100 Sekunden, was etwa zwei Minuten entspricht.
Diese Ergebnisse zeigen also, dass die „adaptiv anisotrope“ Strategie die beste Konvergenzrate aufweist, wir dafür jedoch eine schlechtere Konditionszahl der Matrix in Kauf nehmen müssen. Dies ist letztendlich auf die Größe und Form der Elemente zurückzuführen. An dieser Stelle wollen wir noch zusätzlich Abbildung 8b betrachten, welche das „adaptiv anisotrop“ verfeinerte Netz nach 12 Schritten darstellt. Denn hier erkennen wir sehr gut, dass diese Strategie das Netz insbesondere an den Singularitäten verfeinert.
Ziel wird es nun sein, die Instabilitäten, die bei der analytischen Berechnung auftreten, durch Quadratur zu vermeiden. Dafür werden wir vorher noch Berechnungen mit verschiedenen Quadraturgraden genauer untersuchen.

6.4.2 Vergleich verschiedener Quadraturgrade

Bei der folgenden Berechnung verwenden wir wieder den Algorithmus 6.5 mit Parametern $\theta = 0.5, \nu = 0.5$ zum Lösen von (6.4), wobei alle Integrale von ζ_Q-zulässigen Elementen durch die vorgestellte Gauss-Quadratur approximiert werden. Hierbei wählen wir jeweils 1, 2, 4 oder 8 Auswertungsstellen. Alle vier Berechnungsarten führen wir auf denselben Netzen aus, um die Ergebnisse nicht durch die Wahl des Netzes zu beeinflussen. Zum Berechnen des $\tilde{\mu}_\ell$-Schätzers, welcher die Verfeinerung steuert, verwenden wir in jedem Schritt die Lösung der Quadratur mit 8 Auswertungsstellen.



(a) Fehler $\|\phi - \phi_\ell\|$ und Fehlerschätzer $\tilde{\mu}, \eta$

(b) Berechnungszeit

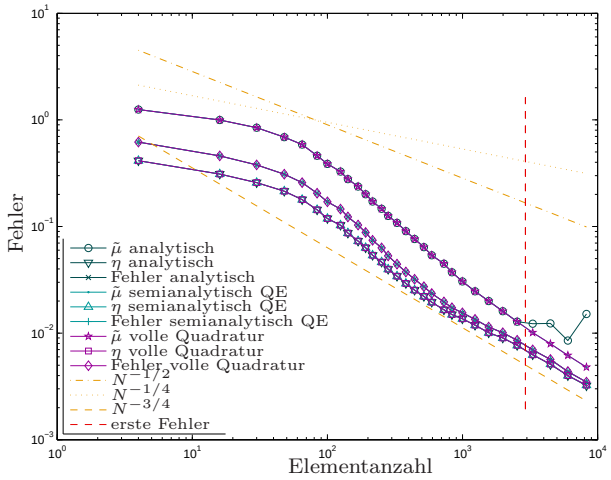
Abbildung 10: Vergleich der Quadraturgrade auf dem Quadrat in Beispiel 6.4.2

In Abbildung 10a haben wir die Ergebnisse der Fehler und Fehlerschätzer für die verschiedenen Quadraturgrade dargestellt. Wir beobachten, dass eine sowie zwei Auswertungsstellen, dargestellt durch die Linien in dunkelgrün und zyan, instabil werden. Für die Quadraturgrade vier und acht, dargestellt durch die Linien in magenta und blau, erkennen wir, dass die Berechnungen stabil bleiben und das auch für Elementanzahlen, bei denen die analytische Berechnung

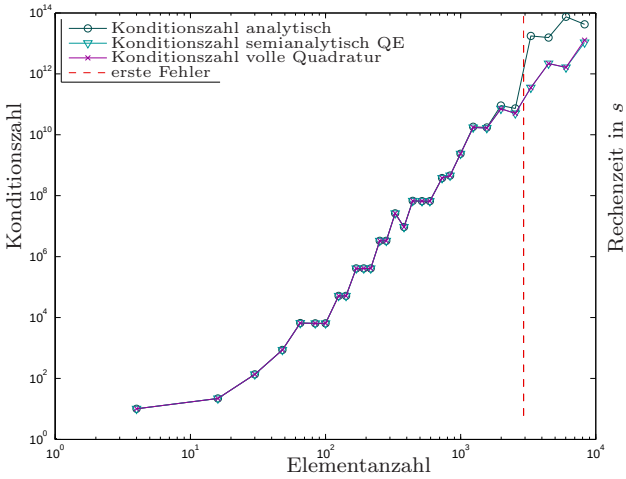
(vergleiche Abbildung 9) versagt.
Da wir möglichst ökonomisch arbeiten wollen, haben wir auch die Berechnungszeiten für die Matrix $\hat{V}_\ell$ in Abbildung 10b untersucht. Wie sich leicht erkennen lässt, benötigt die Berechnung mit Quadraturgrad 8, dargestellt durch die Linie in blau, für 3000 Elemente etwa 1000 Sekunden. Die Berechnungszeiten für die Quadraturgrade 1,2,4 benötigen hingegen wesentlich weniger Rechenzeit und sind fast äquivalent. Sie berechnen die Matrix $\hat{V}_\ell$ mit 3000 Elementen in etwa 100 Sekunden und sind damit etwa 16 Mal schneller als die Quadratur vom Grad 8.
Aufgrund dieser Ergebnisse werden wir für die folgenden Berechnungen einen Quadraturgrad von 4 wählen. Da wir zum einen die Berechnungszeiten gering halten und zum anderen die Stabilität der Berechnungen sicherstellen wollen.

6.4.3 Vergleich verschiedener Berechnungsarten

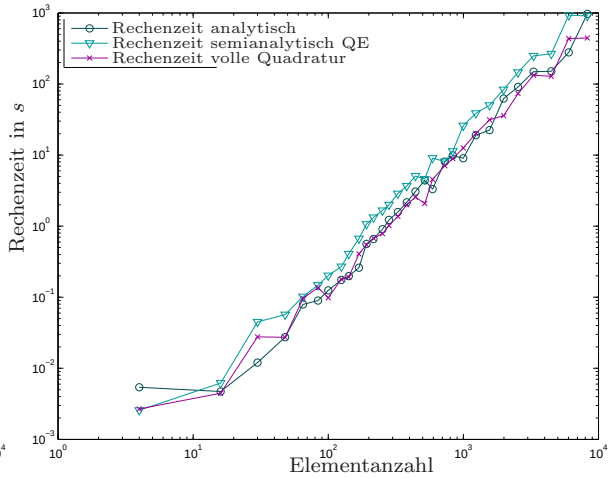
Wie wir für die analytische Berechnung mit adaptiv anisotroper Netzverfeinerung in Abbildung 9a gesehen haben, wird der Fehlerschätzer $\tilde{\mu}_\ell$ ab etwa 3000 Elementen instabil. Deswegen wollen wir an dieser Stelle verschiedene Strategien zur Approximation der Matrix $\hat{V}_\ell$ vorstellen.
Bei den folgenden Berechnungen werden wir wieder den Algorithmus 6.5 mit Parametern $\theta = 0.5, \nu = 0.5$ zum Lösen von (6.4) verwenden, wobei für alle auftretenden Gauss-Quadraturen der Grad 4 gewählt wurde. Alle Berechnungsarten werden auf denselben Netzen ausgeführt um die Ergebnisse nicht durch die Wahl des Netzes zu beeinflussen.
Wir werden nun die Matrix $\hat{V}_\ell$ in der „analytischen“ Strategie durch die in Kapitel 4 vorgestellten Stammfunktionen analytisch berechnen. Dann werden wir für ζ_E-zulässige Elemente wie in Kapitel 3.4 das Integral über das kleinere Element durch Gauss-Quadratur ersetzen. Alle anderen Elemente werden bei dieser „semianalytischen“ Strategie weiterhin analytisch berechnet. In der letzten Strategie „volle Quadratur“ berechnen wir alle ζ_Q-zulässige Elemente wie in Kapitel 3.3 durch Gauss-Quadratur und unzulässige wieder analytisch. Da wir annehmen, dass die letzte Strategie stabil sein wird, werden wir den zugehörigen $\tilde{\mu}_\ell$ Schätzer zum Steuern der Verfeinerung verwenden.
In Abbildung 11a haben wir die Ergebnisse des Fehlers gegenüber der tatsächlichen Lösung und der Fehlerschätzer $\tilde{\mu}, \eta$ für die drei Strategien in Abhängigkeit der Elementanzahl vom Netz $\mathcal{T}_\ell$ eingezeichnet. Anhand der dunkelgrünfarbenen Linie, welche den Fehler und Fehlerschätzer der „analytischen“ Strategie zeigt, beobachten wir wieder eine gute Konvergenzrate bis etwa 2000 Elemente. Ab dieser Grenze, die auch in rot eingetragen wurde, erkennen wir deutlich, dass der $\tilde{\mu}$ Schätzer im Gegensatz zu den anderen Strategien wieder steigt. Der Fehler und η Schätzer bleibt für die „analytische“ Strategie in diesem Vergleich jedoch weiterhin stabil, was auf die Wahl des Steuerparameters der Verfeinerung zurückzuführen ist. An den Linien in zyan und magenta können wir die Ergebnisse für die „semianalytische“ und „volle Quadratur“ Strategie ablesen. Beide Strategien führen zu denselben Ergebnissen und zeigen auch für Berechnungen über 2000 Elemente eine gute Konvergenzrate von $N^{-3/4}$.
In Abbildung 11b vergleichen wir noch einmal die Konditionszahlen der $\hat{V}_\ell$ Matrix für die verschiedenen Strategien. Auch hier wurde die rote Grenzlinie bei etwa 2000 Elementen eingezeichnet. Anhand der Linie in dunkelgrün erkennen wir deutlich, dass ab der 2000 Elemente Grenze die Konditionszahlen für die analytische Berechnung sprunghaft gegenüber den Konditionszahlen der beiden Quadraturen in zyan und magenta ansteigen. Wir erkennen an dieser Stelle aber auch, dass die Kondtionszahlen der „analytischen“ Berechnung schon ab 1000 Elementen von den Quadraturen geringfügig abweichen und damit auf eine ungünstige Berechnung hindeuten.
Weiterhin wollen wir noch einmal die Laufzeit der drei Strategien in Abbildung 11c betrachten. Gemessen wurde die Zeit eines Berechnungsschritts, welches wieder das Aufstellen der Matrizen und Lösen des Gleichungssystems beinhaltet. Hier erkennen wir, dass sich die Laufzeiten der



(a) Fehler $\|\phi - \phi_\ell\|$ und Fehlerschätzer $\tilde{\mu}, \eta$



(b) Konditionszahlen der $\hat{V}_\ell$ Matrix



(c) Berechnungszeit

Abbildung 11: Vergleich der Berechnungsarten von $\hat{V}_\ell$ auf dem Quadrat in Beispiel 6.4.3

drei Strategien nur minimal voneinander unterscheiden. Für fast 10^4 Elemente benötigen alle drei Berechnungsarten etwa 10^3 Sekunden, was etwa 20 Minuten entspricht.
Die Ergebnisse zeigen, dass wir durch Gauss-Quadratur über das kleinere Element auf ζ_E-zulässigen Elementen bis an die Speichergrenze stabil rechnen können und es nicht nötig ist, alle Integrale durch Quadratur zu ersetzen. Weiterhin nimmt die Art der Berechnung von $\hat{V}_\ell$ kaum Einfluss auf die Laufzeit. Damit haben wir in der semianalytischen Berechnung und adaptiv anisotroper Verfeinerung ein stabile Strategie für parallele Elemente gefunden.

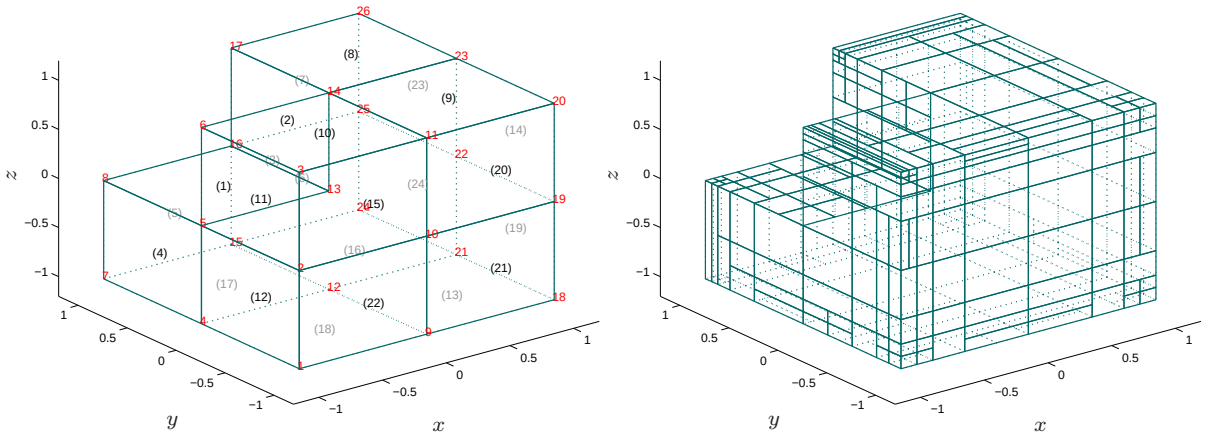
6.5 Beispiel Figuera Würfel

Im Folgenden werden wir die Laplace-Gleichung

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega \subset \mathbb{R}^3, \\ u_{|\Gamma} &= 1 \quad \text{auf } \Gamma, \end{aligned} \tag{6.5}$$

mit dem Startnetz aus Abbildung 12a, einem Figuera Würfel genauer betrachten. Der Figuera Würfel ist gegeben durch einen $[-1,1]^3$ Würfel aus dem ein $[-1,0] \times [0,1]^2$ Würfel heraus

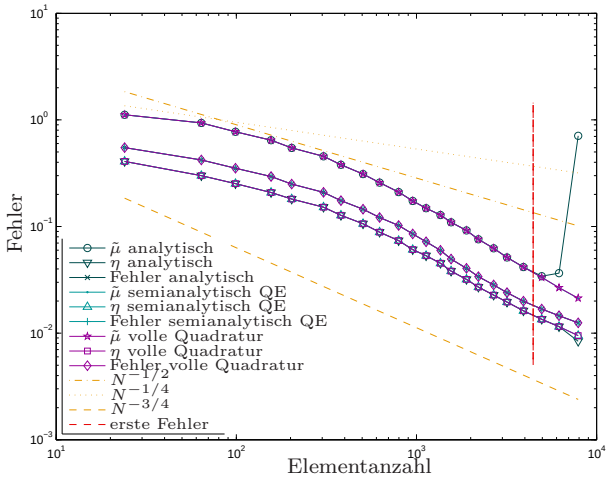
geschnitten wurde. Die Lösung ϕ strebt in diesem Fall an allen außen liegenden Eckpunkten gegen $+\infty$ und an dem innen liegenden Eckpunkt $(0,0,0)$ gegen $-\infty$. Auch hier wurde die Energienorm $\|\phi\|^2 \approx 16.2265$ der exakten Lösung ϕ mithilfe des *Aitken'schen* Δ^2-Verfahrens geschätzt.



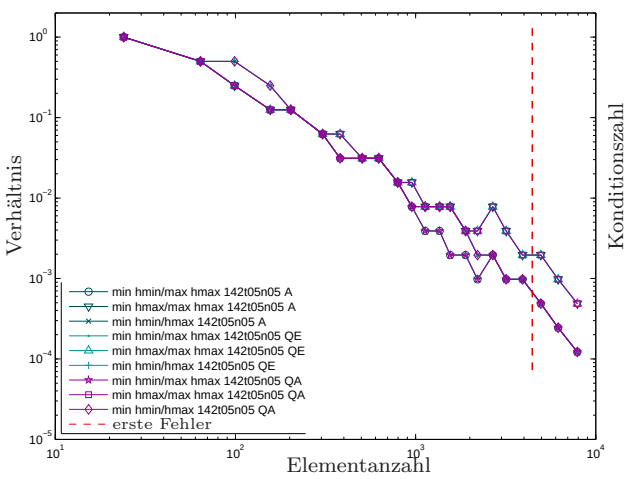
(a) Startnetz (b) Anisotropes Netz nach 6 Schritten mit 381 Elementen

Abbildung 12: Das Startnetz zeigt die Geometrie Figuera Würfel für das Beispiel 6.5. Anhand des Netzes der anisotropen Verfeinerung lässt sich gut erkennen, dass die Geometrie wie erwartet vorwiegend an den Singularitäten, also an den Kanten verfeinert wird.

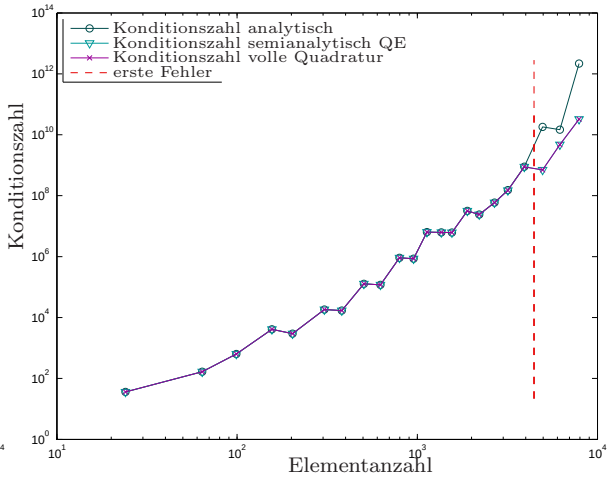
Bei den folgenden Berechnungen werden wir wieder den Algorithmus 6.5 mit Parametern $\theta = 0.5, \nu = 0.5$ zum Lösen von (6.5) verwenden, wobei für alle auftretenden Gauss-Quadraturen der Grad 4 gewählt wurde. Alle Berechnungsarten werden auf denselben Netzen ausgeführt, um die Ergebnisse nicht durch die Wahl des Netzes zu beeinflussen.
Wir werden nun die Matrix $\hat{V}_\ell$ in der „analytischen“ Strategie durch die in Kapitel 4 vorgestellten Stammfunktionen analytisch berechnen. Dann werden wir für ζ_E-zulässige Elemente wie in Kapitel 3.4 das Integral über das kleinere Element durch Gauss-Quadratur ersetzen. Alle anderen Elemente werden bei dieser „semianalytischen“ Strategie weiterhin analytisch berechnet. Weiterhin werden wir in der Strategie „volle Quadratur“ alle auftretenden Integrale für ζ_Q zulässige Elemente durch Gauss-Quadratur ersetzen. Da wir annehmen, dass diese Strategie wieder die zuverlässigste sein wird, werden wir den zugehörigen $\tilde{\mu}$ Schätzer zum Steuern der Netzverfeinerung verwenden.
In Abbildung 13a betrachten wir den Fehler und die Fehlerschätzer $\eta, \tilde{\mu}$ der Galerkin-Lösung für die verschiedenen Strategien. Gerechnet wurde hierbei bis zu einer Elementanzahl von etwa 8000. Anhand der Linien in dunkelgrün erkennen wir für die „analytische“ Berechnung wieder die erwartete Konvergenzrate von $N^{-3/4}$. Da die Kurven des $\tilde{\mu}$ und η Schätzers wieder parallel zum Fehler verlaufen, sind die Fehlerschätzer $\tilde{\mu}$ und η effektiv und zuverlässig. Ab 4000 Elementen beobachten wir am $\tilde{\mu}$ Schätzer, dass die Lösung instabil wird. Diese Grenze haben wir in rot eingezeichnet. Weiterhin erkennen wir anhand der Linien in zyan und magenta, dass die Berechnungen mithilfe von Quadratur weiterhin stabil bleiben und wir dadurch einen Fehler der Energienorm von etwa 0.01 erreichen.
Die Konditionszahlen der $\hat{V}_\ell$ Matrix sehen wir in Abbildung 13c. Bis zu der Elementanzahl 4000 steigt die Kondition der Matrix für alle Strategien auf 10^9 an. Für feinere Netze erkennen wir ab der Grenze von 4000 Elementen wieder den sprunghaften Anstieg der Konditionszahl für die „analytische“ Berechnung in dunkelgrün.



(a) Fehler $\|\phi - \phi_\ell\|$ und Fehlerschätzer $\tilde{\mu}, \eta$



(b) Seitenverhältnisse



(c) Konditionszahlen der $\hat{V}_\ell$ Matrix

Abbildung 13: Vergleich der Berechnungsarten von $\hat{V}_\ell$ auf dem Figuera Würfel in Beispiel 6.5

Diese Beobachtungen zeigen für dreidimensionale Geometrien, dass die analytische Berechnung auf feinen Netzen instabil wird. Vergleichen wir unsere Ergebnisse mit denen aus dem Paper [3, Abbildung 13], so erreichen wir mit der „analytischen“ Strategie die gleiche Genauigkeit. Dass das Netz die Ergebnisse beeinflusst, können wir fast ausschließen, da wir vergleichbare Forderungen und Parameter an die Netzverfeinerung gestellt haben. Durch geeignetes Ersetzen der Integrale durch Gauss-Quadratur können wir jedoch die Genauigkeit der Lösung beliebig erhöhen und erreichen bis zur Netzgröße von 9000 Elementen einen Fehler der Galerkin-Lösung von 0.01. Wenn wir den Fehler für feinere Netze berechnen wollen, müssten wir entweder einen größeren Arbeitsspeicher verwenden oder andere Strategien entwickeln, welche zum Beispiel unter Verwendung von h-Matrizen durch Approximation eine Komprimierung der Matrix zulassen.

A Anhang Code

Abschließend werden wir die für die numerischen Experimente verwendeten Funktionen vorstellen. Hierbei wollen wir anmerken, dass wir im Laufe der Untersuchungen eine Erweiterung eingeführt haben, auf die wir in den vorherigen Kapiteln nicht eingegangen sind.
Um Berechnungsfehler für die Seitenlängen und Flächeninhalte in MATLAB zu vermeiden, haben wir die zusätzliche Matrix $sites \in \mathbb{N}^{N\times 2}$ eingeführt. Eine Zeile $sites_j = (a_j, b_j)$ mit $j \in \{1, \dots, N\}$ der Matrix entspricht der Anzahl der Halbierungen der Seitenlängen a, b aus Definition 2.1 vom Element T_j, gelten muss hierbei $a = 2^{-a_j}$ und $b = 2^{-b_j}$. Dadurch können wir den Flächeninhalt von T_j durch $|T_j| = 2^{-(a_j+b_j)}$ bestimmen.
In der Implementierung für die Berechnung der Matrixeinträge $A_{jk} = \int_{T_j}\int_{T_k} \frac{1}{|\boldsymbol{x}-\boldsymbol{y}|} ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}$ wurde diese Erweiterung nicht verwendet.

A.1 C++

An dieser Stelle werden wir den Quellcode für die Berechnung der Matrix A aus Kapitel 3 und 4 vorstellen.

A.1.1 mex_build_V.cpp

Diese Datei enthält die Implementierung für die Berechnung der Matrix A in MEX. Sie benötigt zum Kompilieren die Dateien `slpRectangle.hpp`, `slpRectangle.cpp` und `gauss.hpp`.
Innerhalb von MATLAB kann die Funktion aufgerufen werden mit

```
[A] = mex_build_V(coo,ele,zeta,typ);
```

Hierbei ist `coo` $\in \mathbb{R}^{M\times 3}$ die Koordinatenmatrix aus Kapitel 5.1 und `ele` $\in \mathbb{N}^{N\times 4}$ die zugehörige Elementmatrix, in der die Zeilenindizes der Koordinaten `coo` stehen. Eine Zeile der Koordinatenmatrix entspricht einer Koordinate und eine Zeile aus der Elementmatrix steht für die Indizes der Koordinaten eines Elements. Weiterhin steht `zeta` $= (p, \zeta_Q, \zeta_E)$ für den Koordinatenvektor. Mithilfe des Parameters $q \in \{0, 1, 2, 3, 4, 5\}$ kann die Anzahl 2^q der Auswertungsstellen für die Gauss-Quadratur gewählt werden. Die beiden Parameter $\zeta_Q, \zeta_E \in \mathbb{R}^+$ werden in den Zulässigkeitsbedingungen verwendet. Der Parameter `typ` $\in \{1, 2, 3\}$ steht für die Art der Berechnung aus Kapitel 5.3. Zurückgegeben wird die zur Parametrisierung gehörende Matrix $A \in \mathbb{R}^{N\times N}$.

```
29 /*
30  * wenn gesetzt wird, paralleles Rechnen benutzt
31  */
32 #define PARALLEL
33 #include <cmath>
34 #include <mex.h>
35
36 #ifdef PARALLEL
37 #include <omp.h>
38 #endif
39
40 #include "slpRectangle.hpp"
41
42 /*
43  * Gibt vom Vektor vec die Dimension != 0 zurueck
44  */
45 int inline dimOfVec(double* vec) {
46   if (vec[2] != 0)
47     return 2;
48   else if (vec[1] != 0)
```

```
49      return 1;
50    else if (vec[0] != 0)
51      return 0;
52    else {
53      mexErrMsgTxt("length of Site is zero");
54      return -1;
55    }
56  }
57
58  /*
59   * Gibt von [0 1 2] die fehlende Zahl zurueck
60   */
61  int inline dimOfThird(int a, int b) {
62    switch (a + b) {
63    case 1:
64      return 2;
65    case 2:
66      return 1;
67    case 3:
68      return 0;
69    default:
70      return -1;
71    }
72  }
73
74  /*
75   * addiert Vektor b zu Vektor a
76   */
77  void inline add(double* a, double* b) {
78    for (int i = 0; i < 3; ++i)
79      a[i] += b[i];
80  }
81
82  /*
83   * vergleicht zwei Vektoren a, b
84   */
85  bool inline iseq(double* a, double* b) {
86    for (int i = 0; i < 3; ++i)
87      if (a[i] != b[i])
88        return false;
89    return true;
90  }
91
92  /*
93   * vergleicht Vektor a mit Konstante b
94   */
95  bool inline iseq(double* a, double b) {
96    for (int i = 0; i < 3; ++i)
97      if (a[i] != b)
98        return false;
99    return true;
100 }
101
102 /*
103  * speichert die Summe der Vektoren a,b in Vektor c
104  */
105 void inline add(double* a, double* b, double* c) {
106   for (int i = 0; i < 3; ++i)
107     a[i] = b[i] + c[i];
108 }
```

```
109
110 /*
111  * subtrahiert Vektor b von Vektor a
112  */
113 void inline sub(double* a, double* b) {
114   for (int i = 0; i < 3; ++i)
115     a[i] -= b[i];
116 }
117
118 /*
119  * speichert die Differenz der Vektoren a,b in Vektor c
120  */
121 void inline sub(double* a, double* b, double* c) {
122   for (int i = 0; i < 3; ++i)
123     a[i] = b[i] - c[i];
124 }
125
126 /*
127  * speichert Vektor b in Vektor a
128  */
129 void inline set(double* a, double* b) {
130   for (int i = 0; i < 3; ++i)
131     a[i] = b[i];
132 }
133
134 /*
135  * setzt alle Werte in Vektor a auf Konstante b
136  */
137 void inline set(double* a, double b) {
138   for (int i = 0; i < 3; ++i)
139     a[i] = b;
140 }
141
142 /*
143  * speichert die kleinste Zeile von Matrix x in Vektor d
144  */
145 void inline getSCorner(double* d, double x[4][3]) {
146   set(d, x[0]);
147   for (int i = 1; i < 4; ++i) {
148     for (int j = 0; j < 3; ++j) {
149       if (d[j] > x[i][j])
150         set(d, x[i]);
151       else if (d[j] < x[i][j])
152         break;
153     }
154   }
155 }
156
157 /*
158  * berechnet den Abstand von zwei parallelen Elementen * 4Pi
159  */
160 double inline distT_par(double b, double d, double t, double v, double d1,
161     double d2, double d3, double * dummy) {
162   double dis1 = 0, dis2 = 0;
163
164   if (d1 < 0)
165     dis1 = -d1 - t;
166   else if (d1 > 0)
167     dis1 = d1 - b;
168   if (dis1 < 0)
```

```
169       dis1 = 0;
170
171     if (d2 < 0)
172       dis2 = -d2 - v;
173     else if (d2 > 0)
174       dis2 = d2 - d;
175     if (dis2 < 0)
176       dis2 = 0;
177
178     return sqrt(dis1 * dis1 + dis2 * dis2 + d3 * d3) * (4 * M_PI);
179   }
180
181   /*
182    * berechnet den Abstand von zwei orthogonalen Elementen * 4Pi
183    */
184   double inline distT_ort(double b, double d, double t, double v, double d1,
185       double d2, double d3, double * dummy) {
186     double dis1 = 0, dis2 = 0, dis3 = 0;
187     if (d1 < 0)
188       dis1 = -d1;
189     else
190       dis1 = d1 - b;
191     if (dis1 < 0)
192       dis1 = 0;
193
194     if (d2 < 0)
195       dis2 = -d2 - t;
196     else
197       dis2 = d2 - d;
198     if (dis2 < 0)
199       dis2 = 0;
200
201     if (d3 < 0)
202       dis3 = -d3 - v;
203     else
204       dis3 = d3;
205     if (dis3 < 0)
206       dis3 = 0;
207
208     return sqrt(dis1 * dis1 + dis2 * dis2 + dis3 * dis3) * (4 * M_PI);
209   }
210
211   /*
212    * mexFunktion zum Aufbauen der Galerkin Matrix
213    */
214   void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
215
216     int i, j, k; //Schleifenindizes
217     bool count = false;
218
219     /*
220      * Sicherheitsueberpruefung der Parameter
221      */
222     if ((nrhs != 4))
223       mexErrMsgTxt(
224           "erwarte (coordinates(Nx3), elements(Mx4), zeta(double), type(int))");
225     if (nlhs > 2 || nlhs == 0)
226       mexErrMsgTxt("Funktion hat maximal 2 Rueckgabewerte [A counts]");
227
228     int cm = mxGetM(prhs[0]);
```

```
229   int cn = mxGetN(prhs[0]);
230   if (cn != 3)
231     mexErrMsgTxt("erwarte Koordinatenmatrix im (Nx3) Format");
232
233   int em = mxGetM(prhs[1]);
234   int en = mxGetN(prhs[1]);
235   if (en != 4)
236     mexErrMsgTxt("erwarte Elementmatrix im (Mx4) Format");
237
238   /*
239    * Parameter fuer parallele Berechnung setzen
240    */
241 #ifdef PARALLEL
242 #define MINSIZE_PER_WORKER 3
243 #define MAX_WORKER 10
244
245   int actualNumberOfThreads = omp_get_max_threads();
246
247   if (MAX_WORKER < actualNumberOfThreads)
248   actualNumberOfThreads = MAX_WORKER;
249
250   int firstRow = 0, lastRow = -1;
251   int targetSize = em * (em + 1) / (2 * actualNumberOfThreads);
252
253   if (targetSize < MINSIZE_PER_WORKER) {
254     actualNumberOfThreads = (int) em * (em + 1) / (2 * MINSIZE_PER_WORKER);
255     if (actualNumberOfThreads < 1) {
256       actualNumberOfThreads = 1;
257       targetSize = em;
258     } else {
259       targetSize = MINSIZE_PER_WORKER;
260     }
261   }
262 #endif
263
264   /*
265    * Variablen initialisieren
266    */
267   plhs[0] = mxCreateDoubleMatrix(em, em, mxREAL);
268
269   double * A = mxGetPr(plhs[0]);
270   double * C = mxGetPr(prhs[0]);
271   double * E = mxGetPr(prhs[1]);
272   double * COUNT = NULL;
273
274   if(nlhs==2){
275     count = true;
276     plhs[1] = mxCreateDoubleMatrix(2,2,mxREAL);
277     COUNT = mxGetPr(plhs[1]);
278   }
279
280   int type = (int) *(mxGetPr(prhs[3]));
281   double * zeta = { 0 };
282
283   if (mxGetN(prhs[2]) > 1)
284     zeta = mxGetPr(prhs[2]) + 1;
285
286   //setzen der Quadraturpunkte
287   int quad = (int) *mxGetPr(prhs[2]);
288   quad = setQuad(quad);
```

```
289
290   //Art der Berechnung bestimmen
291   double (*ctypeP)(double, double, double, double, double, double, double,
292       double*);
293   double (*ctypeO)(double, double, double, double, double, double, double,
294       double*);
295
296   switch (type) {
297   default: // voll analytisch
298     ctypeP = cParO1;
299     ctypeO = cOrtO1;
300     break;
301   case 2: // analytisch oder Qadratur
302     ctypeP = cParO2;
303     ctypeO = cOrtO2;
304     break;
305   case 3: // analytisch oder Qadratur oder Qy[1]x2
306     ctypeP = cParO3;
307     ctypeO = cOrtO3;
308     break;
309   case 4: // analytisch oder Qy[1]x2
310     ctypeP = cParO4;
311     ctypeO = cOrtO4;
312 //    ctypeO = cOrtO2;
313     break;
314   case 0: // Distanz der Elemente
315     ctypeP = distT_par;
316     ctypeO = distT_ort;
317     break;
318   }
319
320   //Lageinformationen
321   int rx, rxa, rxb, ry, rya, ryb;
322   double tmp;
323   double x[4][3] = { { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 } };
324   double y[4][3] = { { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 }, { 0, 0, 0 } };
325   double xa[3] = { 0, 0, 0 };
326   double xb[3] = { 0, 0, 0 };
327   double ya[3] = { 0, 0, 0 };
328   double yb[3] = { 0, 0, 0 };
329   double d[3] = { 0, 0, 0 };
330
331   /*
332    * Schleife ueber Elemente initialisieren
333    */
334 #ifdef PARALLEL
335   {
336 #pragma omp parallel for schedule(static,1)
      private(i,j,k,tmp,x,y,xa,xb,ya,yb,d,rx,rxa,rxb,ry,rya,ryb,lastRow,firstRow)
      shared(C,E,ctypeO,ctypeP,em,targetSize,actualNumberOfThreads)
337
338     for (i = 0; i < actualNumberOfThreads; ++i) {
339       firstRow = (int) sqrt(2 * i * targetSize);
340       if (i == actualNumberOfThreads - 1) {
341         lastRow = em - 1;
342       } else {
343         lastRow = (int) sqrt(2 * (i + 1) * targetSize) - 1;
344       }
345
346       for (j = firstRow; j <= lastRow; ++j) {
```

```
347 #else
348   {
349     {
350       for (j = 0; j < em; ++i) {
351
352 #endif
353         // Eckdaten zwischenspeichern
354         x[0][0] = C[(int) E[j] - 1];
355         x[0][1] = C[cm + (int) E[j] - 1];
356         x[0][2] = C[2 * cm + (int) E[j] - 1];
357
358         x[1][0] = C[(int) E[em + j] - 1];
359         x[1][1] = C[cm + (int) E[em + j] - 1];
360         x[1][2] = C[2 * cm + (int) E[em + j] - 1];
361
362         x[2][0] = C[(int) E[2 * em + j] - 1];
363         x[2][1] = C[cm + (int) E[2 * em + j] - 1];
364         x[2][2] = C[2 * cm + (int) E[2 * em + j] - 1];
365
366         x[3][0] = C[(int) E[3 * em + j] - 1];
367         x[3][1] = C[cm + (int) E[3 * em + j] - 1];
368         x[3][2] = C[2 * cm + (int) E[3 * em + j] - 1];
369
370         // Seitenvektoren aufbauen
371         sub(xa, x[1], x[0]);
372         sub(xb, x[3], x[0]);
373
374         // Lageeigenschaften des Flaechenstuecks
375         rxa = dimOfVec(xa);
376         rxb = dimOfVec(xb);
377         rx = dimOfThird(rxa, rxb);
378
379         if (xa[rxa] <= 0 || xb[rxb] <= 0)
380           mexWarnMsgTxt("X Element hat Seitenlaengen <=0!");
381
382         if (rxa == rxb)
383           mexWarnMsgTxt("X Laenge ist nur in eine Richtung");
384
385         for (k = 0; k <= j; ++k) {
386
387           //Eckdaten zwischenspeichern
388           y[0][0] = C[(int) E[k] - 1];
389           y[0][1] = C[cm + (int) E[k] - 1];
390           y[0][2] = C[2 * cm + (int) E[k] - 1];
391
392           y[1][0] = C[(int) E[em + k] - 1];
393           y[1][1] = C[cm + (int) E[em + k] - 1];
394           y[1][2] = C[2 * cm + (int) E[em + k] - 1];
395
396           y[2][0] = C[(int) E[2 * em + k] - 1];
397           y[2][1] = C[cm + (int) E[2 * em + k] - 1];
398           y[2][2] = C[2 * cm + (int) E[2 * em + k] - 1];
399
400           y[3][0] = C[(int) E[3 * em + k] - 1];
401           y[3][1] = C[cm + (int) E[3 * em + k] - 1];
402           y[3][2] = C[2 * cm + (int) E[3 * em + k] - 1];
403
404           // Seiten Vektoren aufbauen
405           sub(ya, y[1], y[0]);
406           sub(yb, y[3], y[0]);
```

```
407
408             // Lageeigenschaften des Flaechenstuecks
409             rya = dimOfVec(ya);
410             ryb = dimOfVec(yb);
411             ry = dimOfThird(rya, ryb);
412
413             if (ya[rya] <= 0 || yb[ryb] <= 0)
414               mexWarnMsgTxt("Y Element hat Seitenlaengen <=0!");
415
416             if (rya == ryb)
417               mexWarnMsgTxt("Y Laenge ist nur in einer Richtung");
418
419             //delta berechnen
420             sub(d, y[0], x[0]);
421
422             /*
423              * Matrix Eintrag berechnen
424              */
425             if (rx == ry) {
426               if (rxa == rya) {
427                 tmp = ctypeP((xa[rxa]), (xb[rxb]), (ya[rxa]),
428                     (yb[rxb]), d[rxa], d[rxb], d[rx], zeta);
429
430               } else {
431                 tmp = ctypeP((xa[rxa]), (xb[rxb]), (yb[rxa]),
432                     (ya[rxb]), d[rxa], d[rxb], d[rx], zeta);
433               }
434
435             } else {
436
437               if (rxa == rya) {
438                 tmp = ctypeO((xb[rxb]), (xa[rxa]), (ya[rya]),
439                     (yb[ryb]), d[rxb], d[rxa], d[rx], zeta);
440               } else if (rxa == ryb) {
441                 tmp = ctypeO((xb[rxb]), (xa[rxa]), (yb[ryb]),
442                     (ya[rya]), d[rxb], d[rxa], d[rx], zeta);
443               } else if (rxb == rya) {
444                 tmp = ctypeO((xa[rxa]), (xb[rxb]), (ya[rya]),
445                     (yb[ryb]), d[rxa], d[rxb], d[rx], zeta);
446               } else {
447                 tmp = ctypeO((xa[rxa]), (xb[rxb]), (yb[ryb]),
448                     (ya[rya]), d[rxa], d[rxb], d[rx], zeta);
449               }
450
451             }
452 #ifdef PARALLEL
453 #pragma omp critical
454 #endif
455
456             A[(k * em) + j] = 1. / (4 * M_PI) * tmp;
457             if (k != j)
458               A[(j * em) + k] = 1. / (4 * M_PI) * tmp;
459
460 #ifdef PARALLEL
461 #pragma omp end critical
462 #endif
463           }
464         }
465       }
466     }
```

```
467
468   return;
469 }
```

A.1.2 slpRectangle.hpp

Diese Datei stellt die Funktionen zur Berechnung eines Matrixeintrages A_{jk} bereit, welcher dann für parallele Elemente über

```
double cPar01(b,d,t,v,d1,d2,d3,zeta*);
```

berechnet werden kann. `b,d` entspricht den Seitenlängen des Elements T_j und `t,v` denen des Elements T_k und $\boldsymbol{\delta}$ =`(d1,d2,d3)` dem Abstandsvektor, wie in Kapitel 4.1. Hierbei ist zu beachten, dass `b,t` in der selben Dimension liegen, wie auch `d1` und wie auch `d,v,d2` in der selben Dimension liegen. Die Zulässigkeitsbedingungen werden durch `zeta*` $= (\zeta_Q, \zeta_E)$ bestimmt. Für orthogonale Elemente steht analog die Funktion

```
double cOrt01(b,d,t,v,d1,d2,d3,zeta*);
```

zur Verfügung. Hier liegen jedoch, wie in Kapitel 4.1 für orthogonale Elemente vorgestellt, `b,d1` und `d,t,d2` und `v,d3` in jeweils der selben Dimension.

```
13 #ifndef HILBERT3D_LAPLACE_SLPRECTANGLE_HPP_GUARD_
14 #define HILBERT3D_LAPLACE_SLPRECTANGLE_HPP_GUARD_
15
16 /*
17  * voll analytisch rechnen
18  */
19 double cPar01(double, double, double, double, double, double, double, double*);
20 double cOrt01(double, double, double, double, double, double, double, double*);
21
22 /*
23  * volle Quadratur fuer zeta_Q zulaessige Elemente
24  * analytisch sonst
25  */
26 double cPar02(double, double, double, double, double, double, double, double*);
27 double cOrt02(double, double, double, double, double, double, double, double*);
28
29 /*
30  * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
31  * volle Quadratur fuer zeta_Q zulaessige Elemente
32  * analytisch sonst
33  */
34 double cPar03(double, double, double, double, double, double, double, double*);
35 double cOrt03(double, double, double, double, double, double, double, double*);
36
37 /*
38  * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
39  * analytisch sonst
40  */
41 double cPar04(double, double, double, double, double, double, double, double*);
42 double cOrt04(double, double, double, double, double, double, double, double*);
43
44 /*
45  * setzt die 2^Anzahl der Auswertungsstellen
46  */
47 int setQuad(int);
48
49 #endif
```

A.1.3 slpRectangle.cpp

Diese Datei implementiert die Funktionen aus `slpRectangle.hpp`. Hierzu wurden die vorgestellten Funktionen aus Kapitel 4 umgesetzt, wie auch die zugehörigen Quadraturen.

```
13 #include <math.h>
14
15 #include <mex.h>
16
17 #include "slpRectangle.hpp"
18 #include "gauss.hpp"
19
20 int GAUSS_size = GAUSS_SIZE[2];
21 gauss * GAUSS_nodes = GAUSS_NODES[2];
22
23
24
25 /*
26  * setzt den Quadraturgrad, also 2^q Auswertungsstellen
27  */
28 int setQuad(int q) {
29   GAUSS_size = GAUSS_SIZE[q];
30   GAUSS_nodes = GAUSS_NODES[q];
31   return GAUSS_size;
32 }
33
34 /*
35  * gibt das Signum von x zurueck
36  */
37 int inline sign(double x) {
38   return x > 0 ? 1 : (x < 0 ? -1 : 0);
39 }
40
41 /*
42  * gibt das Maximum der beiden Zahlen x,y zurueck
43  */
44 double inline max(double x, double y) {
45   return x < y ? y : x;
46 }
47
48 /*
49  * gibt das Minimum der beiden Zahlen x,y zurueck
50  */
51 double inline min(double x, double y) {
52   return x > y ? y : x;
53 }
54
55 /*
56  * kleinere Seiten nach vorn und dadurch kleinerer Durchmesser nach vorn
57  */
58 void inline switch_site_par(double& b, double& d, double& t, double& v,
59     double& d1, double& d2) {
60   double tmp = 0;
61
62   // kleine Seite nach vorn (x1 y1)
63   if (b > t) {
64     tmp = b;
65     b = t;
66     t = tmp;
67     d1 = -d1;
68   }
```

```
69
70    // kleine Seite nach vorn (x2 y2)
71    if (d > v) {
72      tmp = d;
73      d = v;
74      v = tmp;
75      d2 = -d2;
76    }
77  }
78
79  /*
80   * kleinere Seiten nach vorn und dadurch kleinerer Durchmesser nach vorn
81   */
82  void inline switch_site_ort(double& b, double& d, double& t, double& v,
83      double& d1, double& d2, double& d3) {
84    double tmp = 0;
85
86    // kleine Seite nach vorn (x2 y2)
87    if (d > t) {
88      tmp = d;
89      d = t;
90      t = tmp;
91      d2 = -d2;
92    }
93
94    // kleine Seite nach vorn (x1 y3)
95    if (b > v) {
96      tmp = b;
97      b = v;
98      v = tmp;
99      tmp = -d1;
100     d1 = -d3;
101     d3 = tmp;
102   }
103 }
104
105 /*
106  * kleinere Achse nach vorn
107  */
108 void inline switch_dim_par(double& b, double& d, double& t, double& v,
109     double& d1, double& d2) {
110   //
111   if (b * b + t * t > d * d + v * v) {
112     double tmp = 0;
113
114     tmp = b;
115     b = d;
116     d = tmp;
117     tmp = t;
118     t = v;
119     v = tmp;
120
121     tmp = d1;
122     d1 = d2;
123     d2 = tmp;
124   }
125 }
126
127 /*
128  * gibt den Abstand zum Quadrat zurueck
```

```
129  */
130 double inline dist2_par(double b, double d, double t, double v, double d1,
131     double d2, double d3) {
132   double dis1 = 0, dis2 = 0;
133   if (d1 < 0)
134     dis1 = -d1 - t;
135   else
136     dis1 = d1 - b;
137   if (dis1 < 0)
138     dis1 = 0;
139 
140   if (d2 < 0)
141     dis2 = -d2 - v;
142   else
143     dis2 = d2 - d;
144   if (dis2 < 0)
145     dis2 = 0;
146 
147   return dis1 * dis1 + dis2 * dis2 + d3 * d3;
148 }
149 
150 /*
151  * gibt den Abstand zum Quadrat zurueck
152  */
153 double inline dist2_ort(double b, double d, double t, double v, double d1,
154     double d2, double d3) {
155   double dis1 = 0, dis2 = 0, dis3 = 0;
156   if (d1 < 0)
157     dis1 = -d1;
158   else
159     dis1 = d1 - b;
160   if (dis1 < 0)
161     dis1 = 0;
162 
163   if (d2 < 0)
164     dis2 = -d2 - t;
165   else
166     dis2 = d2 - d;
167   if (dis2 < 0)
168     dis2 = 0;
169 
170   if (d3 < 0)
171     dis3 = -d3 - v;
172   else
173     dis3 = d3;
174   if (dis3 < 0)
175     dis3 = 0;
176 
177   return dis1 * dis1 + dis2 * dis2 + dis3 * dis3;
178 }
179 
180 /*
181  * gibt den Abstand in einer Richtung zum Quadrat zurueck
182  */
183 double inline dist_s2(double b, double t, double d1) {
184   double dis1 = 0;
185   if (d1 < 0)
186     dis1 = -d1 - t;
187   else
188     dis1 = d1 - b;
```

```
189   if (dis1 < 0)
190     dis1 = 0;
191
192   return dis1 * dis1;
193 }
194
195 /*
196  * Kernfunktion 1 /|x-y|
197  */
198 double inline f_par(double x1, double x2, double y1, double y2, double d1,
199     double d2, double d3) {
200   return 1.
201       / sqrt(
202           (x1 - y1 - d1) * (x1 - y1 - d1)
203               + (x2 - y2 - d2) * (x2 - y2 - d2) + d3 * d3);
204 }
205 /*
206  * Kernfunktion 1 /|x-y|
207  */
208 double inline f_ort(double x1, double x2, double y2, double y3, double d1,
209     double d2, double d3) {
210   return 1.
211       / sqrt(
212           (x1 - d1) * (x1 - d1) + (x2 - y2 - d2) * (x2 - y2 - d2)
213               + (y3 + d3) * (y3 + d3));
214 }
215
216 /*
217  * 1 mal integrierte Kernfunktion int 1 /|x-y| dy (QUADRATUR)
218  */
219 double inline g_QY(double p, double y, double x, double l) {
220   double sol = 0;
221   for (int i = 0; i < GAUSS_size; ++i) {
222     sol += pow(
223         (x - GAUSS_nodes[i].n * y) * (x - GAUSS_nodes[i].n * y) + l * l,
224         p) * GAUSS_nodes[i].w * y;
225   }
226
227   return sol;
228 }
229
230 /*
231  * Stammfunktion g
232  * 1 mal integrierte Kernfunktion int 1 /|x-y| dy (ANALYTISCH)
233  */
234 double inline g_AY(double p, double y, double x, double l) {
235   double sol = 0;
236
237   if (l != 0) {
238     if (p == 0.5) {
239       sol = (y - x) / 2 * sqrt((y - x) * (y - x) + l * l)
240           + l * l / 2 * asinh((y - x) / fabs(l));
241     } else if (p == 0)
242       sol = y - x;
243     else if (p == -0.5)
244       sol = asinh((y - x) / fabs(l));
245     else if (p == -1)
246       sol = atan((y - x) / fabs(l)) / fabs(l);
247     else if (p == -1.5)
248       sol = (y - x) / ((l * l) * sqrt((y - x) * (y - x) + l * l));
```

```
249       else
250         sol = (y - x) * pow((y - x) * (y - x) + l * l, p)
251             + 2 * p * l * l * g_AY(p - 1, y, x, l) / (2 * p + 1);
252     } else {
253       if (p == -0.5)
254         sol = sign(y - x) * log(fabs(y - x));
255       else
256         sol = (y - x) * pow(fabs(y - x), 2 * p) / (2 * p + 1);
257     }
258
259     return sol;
260   }
261
262   /*
263    * 2 mal integrierte Kernfunktion int int 1 /|x-y| dy_2 dy_1  (QUADRATUR)
264    */
265   double inline G_QY1Y2(double p, double y1, double y2, double x1, double x2,
266       double l) {
267
268     double sol = 0;
269     for (int i = 0; i < GAUSS_size; ++i) {
270       for (int j = 0; j < GAUSS_size; ++j) {
271         sol += pow(
272             (x1 - y1 * GAUSS_nodes[i].n) * (x1 - y1 * GAUSS_nodes[i].n)
273                 + (x2 - y2 * GAUSS_nodes[j].n)
274                     * (x2 - y2 * GAUSS_nodes[j].n) + l * l, p)
275             * y1 * GAUSS_nodes[i].w * y2 * GAUSS_nodes[j].w;
276       }
277     }
278     return sol;
279   }
280
281   /*
282    * 2 mal integrierte Kernfunktion int int 1 /|x-y| dy_2 dx_2  (ANALYTISCH)
283    */
284   double inline G_AY2X2(double y1, double y2, double x1, double x2, double d3) {
285     double hlp = ((x1 - y1) * (x1 - y1) + (x2 - y2) * (x2 - y2) + d3 * d3);
286     double sol = sqrt(hlp);
287
288     if ((x2 - y2) != 0)
289       sol += (x2 - y2) * log(sqrt(hlp) - (x2 - y2));
290
291     return sol;
292   }
293
294   /*
295    * Stammfunktion G
296    * 2 mal integrierte Kernfunktion int int 1 /|x-y| dy_2 dy_1  (ANALYTISCH)
297    */
298   double inline G_AY1Y2(double p, double y1, double y2, double x1, double x2,
299       double l) {
300     double pt = p + 1.5;
301     double sol = 0;
302     if (pt == 0) {
303       if (l == 0) {
304         sol = -sqrt((y1 - x1) * (y1 - x1) + (y2 - x2) * (y2 - x2))
305             / ((y1 - x1) * (y2 - x2));
306       } else {
307         sol = sign((y1 - x1) * (y2 - x2)) / (2 * fabs(l))
308             * acos(
```

```
                -2 * (y1 - x1) * (y1 - x1) * (y2 - x2) * (y2 - x2)
                    / (((y1 - x1) * (y1 - x1) + l * l)
                        * ((y2 - x2) * (y2 - x2) + l * l))
                    + 1);
    }
  } else if ((pt > 0) && ((int) pt == pt)) {
    if (l != 0)
      sol = 2 * p * l * l * G_AY1Y2(p - 1, y1, y2, x1, x2, l);
    if ((y1 - x1) != 0)
      sol += (y1 - x1)
          * g_AY(p, y2, x2, sqrt((y1 - x1) * (y1 - x1) + l * l));
    if ((y2 - x2) != 0)
      sol += (y2 - x2)
          * g_AY(p, y1, x1, sqrt((y2 - x2) * (y2 - x2) + l * l));
    sol /= 2 * p + 2;
  } else {
    //Fall wird nicht benoetigt
    sol = NAN;
    mexWarnMsgTxt("G_AY1Y2 fuer Parameter nicht implementiert!");
  }

  return sol;
}

/*
 * Stammfunktion F
 * 4 mal integrierte Kernfunktion int int 1 /|x-y| dy dx  (ANALYTISCH)
 */
double inline F_par(double x1, double x2, double y1, double y2, double d1,
    double d2, double d3) {
  double sol = (x1 - y1 - d1) * (x2 - y2 - d2);

  if (sol != 0)
    sol *= G_AY1Y2(-0.5, x1, x2, y1 + d1, y2 + d2, d3);

  if ((x1 - y1 - d1) != 0)
    sol -= (x1 - y1 - d1)
        * g_AY(0.5, x1, y1 + d1,
            sqrt((x2 - y2 - d2) * (x2 - y2 - d2) + d3 * d3));

  if ((x2 - y2 - d2) != 0)
    sol -= (x2 - y2 - d2)
        * g_AY(0.5, x2, y2 + d2,
            sqrt((x1 - y1 - d1) * (x1 - y1 - d1) + d3 * d3));

  double hlp = ((x1 - y1 - d1) * (x1 - y1 - d1)
      + (x2 - y2 - d2) * (x2 - y2 - d2) + d3 * d3);
  sol += 1. / 3 * hlp * sqrt(hlp);
  return sol;
}

/*
 * Stammfunktion F
 * 4 mal integrierte Kernfunktion int int 1 /|x-y| dy dx  (ANALYTISCH)
 */
double inline F_ort(double x1, double x2, double y2, double y3, double d1,
    double d2, double d3) {
  double sol = -G_AY1Y2(0.5, y3, x1, -d3, d1, x2 - y2 - d2);

  if ((x1 - d1) * (x2 - y2 - d2) != 0)
```

```
    sol -= (x1 - d1) * (x2 - y2 - d2)
        * G_AY1Y2(-0.5, x2, y3, y2 + d2, -d3, x1 - d1);

  if ((x1 - d1) != 0)
    sol += (x1 - d1)
        * g_AY(0.5, y3, -d3,
            sqrt(
                (x1 - d1) * (x1 - d1)
                    + (x2 - y2 - d2) * (x2 - y2 - d2)));

  if ((y3 + d3) * (x2 - y2 - d2) != 0)
    sol -= (y3 + d3) * (x2 - y2 - d2)
        * G_AY1Y2(-0.5, x1, x2, d1, y2 + d2, -y3 - d3);

  if ((y3 + d3) != 0)
    sol += (y3 + d3)
        * g_AY(0.5, x1, d1,
            sqrt(
                (x2 - y2 - d2) * (x2 - y2 - d2)
                    + (y3 + d3) * (y3 + d3)));

  return sol / 2.;
}

/*
 * Stammfunktion (fuer Quadratur)
 * 2 mal integrierte Kernfunktion int int 1 /|x-y| dy_2 dy_1  (ANALYTISCH)
 */
double inline FY1Y2_par(double x1, double x2, double y1, double y2, double d1,
    double d2, double d3) {
  return G_AY1Y2(-0.5, y1, y2, x1 - d1, x2 - d2, d3);
}

/*
 * Stammfunktion (fuer Quadratur)
 * 2 mal integrierte Kernfunktion int int 1 /|x-y| dy_2 dy_3  (ANALYTISCH)
 */
double inline FY2Y3_ort(double x1, double x2, double y2, double y3, double d1,
    double d2, double d3) {
  return G_AY1Y2(-0.5, y2, y3, x2 - d2, -d3, x1 - d1);
}

/*
 * Quadratur ueber zwei Integrale und Grenzen in Stammfunktion ueber zwei Integrale
 * b,d - Grenzen fuer Quadratur
 * t,v - Grenzen in Stammfunktion
 */
template<double (F)(double, double, double, double, double, double, double)>
double intQ2A2(double b, double d, double t, double v, double d1, double d2,
    double d3) {
  double sol = 0;
  double w1, x1, x2;

  for (int i = 0; i < GAUSS_size; ++i) {
    x1 = b * GAUSS_nodes[i].n;
    w1 = GAUSS_nodes[i].w;
    for (int j = 0; j < GAUSS_size; ++j) {
      x2 = d * GAUSS_nodes[j].n;
      sol += (F(x1, x2, t, v, d1, d2, d3) - F(x1, x2, 0, v, d1, d2, d3)
          - F(x1, x2, t, 0, d1, d2, d3) + F(x1, x2, 0, 0, d1, d2, d3))
```

```
429             * w1 * GAUSS_nodes[j].w;
430       }
431     }
432
433     return sol * b * d;
434   }
435
436   /*
437    * Quadratur ueber vier Integrale
438    * b,d,t,v - Grenzen fuer Quadratur
439    */
440   template<double (F)(double, double, double, double, double, double, double)>
441   double intQ4(double b, double d, double t, double v, double d1, double d2,
442       double d3) {
443
444     double sol = 0;
445     int i, j, k, l;
446     double x1, x2, y1;
447     double w1, w2, w3;
448
449     for (i = 0; i < GAUSS_size; ++i) {
450       x1 = b * GAUSS_nodes[i].n;
451       w1 = GAUSS_nodes[i].w;
452       for (j = 0; j < GAUSS_size; ++j) {
453         x2 = d * GAUSS_nodes[j].n;
454         w2 = w1 * GAUSS_nodes[j].w;
455         for (k = 0; k < GAUSS_size; ++k) {
456           y1 = t * GAUSS_nodes[k].n;
457           w3 = w2 * GAUSS_nodes[k].w;
458           for (l = 0; l < GAUSS_size; ++l) {
459             sol += F(x1, x2, y1, v * GAUSS_nodes[l].n, d1, d2, d3) * w3
460                 * GAUSS_nodes[l].w;
461           }
462         }
463       }
464     }
465
466     return sol * b * d * t * v;
467   }
468
469   /*
470    * Grenzen in Stammfunktion ueber vier Integrale
471    * b,d,t,v - Grenzen fuer Stammfunktion
472    */
473   template<double (F)(double, double, double, double, double, double, double)>
474   double inline intA4(double b, double d, double t, double v, double d1,
475       double d2, double d3) {
476
477     return F(b, d, t, v, d1, d2, d3) - F(b, d, t, 0, d1, d2, d3)
478         - F(b, d, 0, v, d1, d2, d3) + F(b, d, 0, 0, d1, d2, d3)
479         - F(b, 0, t, v, d1, d2, d3) + F(b, 0, t, 0, d1, d2, d3)
480         + F(b, 0, 0, v, d1, d2, d3) - F(b, 0, 0, 0, d1, d2, d3)
481         - F(0, d, t, v, d1, d2, d3) + F(0, d, t, 0, d1, d2, d3)
482         + F(0, d, 0, v, d1, d2, d3) - F(0, d, 0, 0, d1, d2, d3)
483         + F(0, 0, t, v, d1, d2, d3) - F(0, 0, t, 0, d1, d2, d3)
484         - F(0, 0, 0, v, d1, d2, d3) + F(0, 0, 0, 0, d1, d2, d3);
485
486   }
487
488   /*
```

```
 * fuer parallele Elemente
 * voll analytisch
 */
double cParO1(double b, double d, double t, double v, double d1, double d2,
    double d3, double* zeta) {
  return intA4<F_par>(b, d, t, v, d1, d2, d3);
}
/*
 * fuer orthogonale Elemente
 * voll analytisch
 */
double cOrtO1(double b, double d, double t, double v, double d1, double d2,
    double d3, double* zeta) {
  return intA4<F_ort>(b, d, t, v, d1, d2, d3);
}

/*
 * fuer parallele Elemente
 * volle Quadratur fuer zeta_Q zulaessige Elemente
 * analytisch sonst
 */
double cParO2(double b, double d, double t, double v, double d1, double d2,
    double d3, double* zeta) {
  double tmp = 0;

  //kurze Seite nach vorn
  switch_site_par(b, d, t, v, d1, d2);

  if (zeta[0] * zeta[0] * (t * t + v * v) < dist2_par(b, d, t, v, d1, d2, d3))
    return intQ4<f_par>(b, d, t, v, d1, d2, d3);

  return intA4<F_par>(b, d, t, v, d1, d2, d3);
}

/*
 * fuer orthogonale Elemente
 * volle Quadratur fuer zeta_Q zulaessige Elemente
 * analytisch sonst
 */
double cOrtO2(double b, double d, double t, double v, double d1, double d2,
    double d3, double* zeta) {
  double tmp = 0, dis2 = 0;

  //kurze Seite nach vorn
  switch_site_ort(b, d, t, v, d1, d2, d3);

  if (zeta[0] * zeta[0] * (t * t + v * v) < dist2_ort(b, d, t, v, d1, d2, d3))
    return intQ4<f_ort>(b, d, t, v, d1, d2, d3);

  return intA4<F_ort>(b, d, t, v, d1, d2, d3);
}

/*
 * fuer parallele Elemente
 * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
 * volle Quadratur fuer zeta_Q zulaessige Elemente
 * analytisch sonst
 */
double cParO3(double b, double d, double t, double v, double d1, double d2,
    double d3, double* zeta) {
```

```
549
550   //kurze Seite nach vorn
551   switch_site_par(b, d, t, v, d1, d2);
552
553   if (zeta[1] * zeta[1] * (b * b + d * d) < dist2_par(b, d, t, v, d1, d2, d3))
554     return intQ2A2<FY1Y2_par>(b, d, t, v, d1, d2, d3);
555
556   if (zeta[0] * zeta[0] * (t * t + v * v) < dist2_par(b, d, t, v, d1, d2, d3))
557     return intQ4<f_par>(b, d, t, v, d1, d2, d3);
558
559   return intA4<F_par>(b, d, t, v, d1, d2, d3);
560 }
561
562 /*
563  * fuer orthogonale Elemente
564  * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
565  * volle Quadratur fuer zeta_Q zulaessige Elemente
566  * analytisch sonst
567  */
568 double cOrt03(double b, double d, double t, double v, double d1, double d2,
569     double d3, double* zeta) {
570   double tmp = 0, dis2 = 0;
571
572   //kurze Seite nach vorn
573   switch_site_ort(b, d, t, v, d1, d2, d3);
574
575   if (zeta[1] * zeta[1] * (b * b + d * d) < dist2_ort(b, d, t, v, d1, d2, d3))
576     return intQ2A2<FY2Y3_ort>(b, d, t, v, d1, d2, d3);
577
578   if (zeta[0] * zeta[0] * (t * t + v * v) < dist2_ort(b, d, t, v, d1, d2, d3))
579     return intQ4<f_ort>(b, d, t, v, d1, d2, d3);
580
581   return intA4<F_ort>(b, d, t, v, d1, d2, d3);
582 }
583
584 /*
585  * fuer parallele Elemente
586  * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
587  * analytisch sonst
588  */
589 double cPar04(double b, double d, double t, double v, double d1, double d2,
590     double d3, double* zeta) {
591
592   //kurze Seite nach vorn
593   switch_site_par(b, d, t, v, d1, d2);
594
595   if (zeta[1] * zeta[1] * (b * b + d * d) < dist2_par(b, d, t, v, d1, d2, d3))
596     return intQ2A2<FY1Y2_par>(b, d, t, v, d1, d2, d3);
597
598   return intA4<F_par>(b, d, t, v, d1, d2, d3);
599 }
600 /*
601  * fuer orthogonale Elemente
602  * Quadratur ueber ein Element fuer zeta_E zulaessige Elemente
603  * analytisch sonst
604  */
605 double cOrt04(double b, double d, double t, double v, double d1, double d2,
606     double d3, double* zeta) {
607   double tmp = 0, dis2 = 0;
608
```

```
609   //kurze Seite nach vorn
610   switch_site_ort(b, d, t, v, d1, d2, d3);
611
612   if (zeta[1] * zeta[1] * (b * b + d * d) < dist2_ort(b, d, t, v, d1, d2, d3))
613     return intQ2A2<FY2Y3_ort>(b, d, t, v, d1, d2, d3);
614
615   return intA4<F_ort>(b, d, t, v, d1, d2, d3);
616 }
```

A.1.4 gauss.hpp

Diese Datei wurde automatisch von der MATLAB-Funktion `export_gauss.m` erstellt. Durch den Befehl

```
export_gauss(start,stop,steps,file);
```

wird die Datei `file` angelegt, welche dann in C++verwendet werden kann. In ihr sind die Gauss-Quadraturpunkte und Gewichte für das Intervall $[start, stop]$ gespeichert. `steps` steht hierbei für einen Vektor, welcher die zur Verfügung stehenden Grade steuert.
In C++ kann die Quadratur dann wie im folgenden Beispiel mit Quadraturgrad 2^3 benutzt werden.

```
1  double sol;
2  for(int i=0;i<GAUSS_SIZE[3];++i)
3    sol += sin(GAUSS_NODES[3][i].n) * GAUSS_NODES[3][i].w;
```

Da die Datei fast nur Quadraturpunkte und Gewichte enthält, wollen wir hier nur den Anfang vorstellen.

```
26 #ifndef GAUSS_NODES_
27 #define GAUSS_NODES_
28
29 typedef struct _gauss {
30   double n;
31   double w;
32 } gauss;
33
34 double GAUSS_SIZE[] = { 1, 2, 4, 8, 16, 32 };
35
36 gauss GAUSS_NODES[][32] = {
37 {
38   { 0.5, 1 }},
39 {
40   { 0.21132486540518707895, 0.49999999999999888978 },
41   { 0.78867513459481286529, 0.49999999999999888978 }},
```

A.2 MATLAB

Hier wollen wir alle MATLAB-Funktionen für den adaptiven Algorithmus 6.5 vorstellen. Funktionen, die lediglich zum Erstellen von Grafiken und zum Testen dienen, werden wir hier nicht zeigen.

A.2.1 compute.m

Diese Funktion implementiert im Wesentlichen den Algorithmus 6.5. Nachdem das Startnetz aus der Datei `file` geladen wurde, werden `times` $\in \mathbb{N}$ Verfeinerungsschritte durchgeführt. Sollte `times` größer als 40 sein, wird nicht nach 40 Schritten abgebrochen, sondern nachdem die Elementanzahl `times` vom Netz $\mathcal{T}_\ell$ erreicht wurde. Die Berechnungsarten werden über den Vektor

typ $\in \{1,2,3,4\}^n$ bestimmt. Hierbei werden in jedem Verfeinerungsschritt alle n Berechnungsarten aus typ auf dem selben Netz durchgeführt. zeta ist ein Array, welcher zu den n Berechnungsarten den entsprechenden $\zeta = (p, \zeta_Q, \zeta_E)$ Vektor enthält. Zum Steuern des Verfeinern-Algorithmus 2.8 dienen die Parameter theta,nu $\in [0,1]$ und über den Parameter vcon $\in \{0,1\}$ kann die Vorkonditionierung der A Matrix eingeschaltet werden. Das neue Netz $\mathcal{T}_{\ell+1}$ wird durch die letzte Berechnungsart bestimmt. Zurückgegeben wird die Matrix data, in der alle wichtigen Auswertungen gespeichert sind. Die folgende Zeile

```
compute('exmpl_2DQuad',500,{[0 0 0] [3 2 2]},[1 2],0.5,0.5,0);
```

führt eine adaptiv anisotrope ($\theta = 0.5, \nu = 0.5$) voll analytische und volle Quadratur für $\zeta_Q = 2$-zulässige Elemente, (typ=[1 2]) Berechnung bis times=500 Elemente auf einem Quadrat durch. Hierbei wird eine Gauss-Quadratur vom Grad 8 verwendet, keine Vorkonditionierung und zum Markieren der zu verfeinernden Elemente die Galerkin-Lösung zum Typ „volle Quadratur“. Gelöst wird das Problem aus (6.4).

```
1  function [data er fileo] = compute(file,times,zeta,typ,theta,nu,vcon)

19 % Datei laden
20 load(file)
21
22 % Container fuer Messungen initialisieren
23 if(~exist('data','var'))
24     data=[];
25 end
26
27 kap3 = 0;
28
29 tic
30
31 %times mal verfeinern oder bis Elementanzahl fuer times > 40
32 for j = 1:times
33   %beende Schleife wenn Elementanzahl erreicht
34   if(times>40 && size(elements,1) > times)
35     break;
36   end
37
38   %altes Netz mit Loesung merken
39   if(exist('coo_fine','var'))
40     old_C_fine = coo_fine;
41     old_F_fine = f2s;
42     old_E_fine = ele_fine;
43     old_x_fine = x_fine;
44   end
45
46   %uniformIsotrop verfeinern
47   [coo_fine,ele_fine,neigh_fine,f2s,sit_fine]...
48     =refineQuad(coordinates,elements,neigh,sites,2);
49
50   %Flaecheninhalte berechnen (rhs)
51   b_fine = 2.^-sum(sit_fine,2);
52
53   b = 2.^-sum(sites,2);
54   hmin = 2.^-max(sites,[],2);
55   hmax = 2.^-min(sites,[],2);
56
57
58   %data -> ErgebnisVektor
59   dataS = size(elements,1);
60
```

```
61    %zeta vorbereiten
62    if(~iscell(zeta))
63      zeta_tmp = zeta;
64      clear zeta;
65      for i = 1:length(typ)
66        zeta{i} = zeta_tmp;
67      end
68    end
69
70    %alle MatrixBrechenungsArten mit dem selben Netz berechnen
71    for i = 1:length(typ)
72      start_time = toc;
73      disp([ '[' num2str(j) ',' num2str(i) '] ' ...
74        num2str(size(elements,1)) ' : ' t2str(toc) ' ->' num2str(typ(i))])
75      %Matrix aufbauen -> MEX
76      V_fine = mex_build_V(coo_fine,ele_fine,zeta{i},typ(i));
77      build_time = toc;
78      %testet auf fehlerhafte Eintraege (NaN +/-Inf)
79      [r c] = find(isnan(V_fine)~=isinf(V_fine));
80      if(~isempty(r))
81          figure(9)
82          plotShape(coo_fine,ele_fine(unique([r c]),:),'')
83          plotMark([r';c'],coo_fine,ele_fine)
84          title('Fehlerhafte Elemente')
85      end
86
87      if(~vcon)
88      %Loesung berechnen
89          x_fine = V_fine\b_fine;
90
91      else
92      %Vorkonditionierte Loesung!
93          D = diag(V_fine).^(-1/2);
94          for k = 1:length(V_fine)
95            for l = 1:length(V_fine)
96              V_fine(k,l) = V_fine(k,l)*D(k)*D(l);
97            end
98          end
99          c = D.*b_fine;
100         y = V_fine\c;
101         x_fine = D.*y;
102     end
103
104     solve_time = toc;
105
106     %Konditionszahl aufstellen
107     con = cond(V_fine);
108
109     clear V_fine
110
111     % \tilde \mu ( \Rho h -h + L_2 )
112     tmu = hmin.* b .* ...
113       sum((x_fine(f2s)'-repmat(sum(x_fine(f2s)',1)/4,4,1)).^2)' /4;
114
115     %Fehlerschaetzer 2 aufbauen
116     V = mex_build_V(coordinates,elements,zeta{i},typ(i));
117
118     if(~vcon)
119         x = V\b;
120     else
```

```
121         D = diag(V).^(-1/2);
122         for k = 1:length(V)
123           for l = 1:length(V)
124             V(k,l) = V(k,l)*D(k)*D(l);
125           end
126         end
127         c = D.*b;
128         y = V\c;
129         x = D.*y;
130       end
131
132       clear V
133
134       xo_fine(f2s) = repmat(x,1,4);
135       xd_fine = abs(xo_fine'-x_fine);
136
137       % \mu ( h/2 -h + L_2 )
138       mu = hmin.*b.*sum((x_fine(f2s)'-repmat(x',4,1)).^2)'/4;
139
140
141       %Energienorm^2 berechnen |||h||| & |||h/2|||
142   %     xe_fine = x_fine'*A_fine*x_fine;
143       xe_fine = b_fine'*x_fine;
144   %     xe = x'*A*x;
145       xe = b'*x;
146
147
148       %Enorm^2 Elementweise vergleichen
149       if(exist('old_F_fine','var'))
150         for k = 1:size(ele_fine,1)
151           e_f = find(sum(k==f2s,2));
152           e_f_p = find(f2s(e_f,:)==k);
153           assert(size(e_f_p,1) ==1,'Error: e_f_p wurde mehrfach gefunden')
154           e_ff = find(sum(e_f==f,2));
155           e_ff_p = find(f(e_ff,:)==e_f);
156           assert(size(e_ff_p,1)==1,'Error: e_ff_p wurde mehrfach gefunden')
157           e_ff_s = length(e_ff_p);
158           if e_ff_s==4
159              e_p = e_f_p;
160           elseif e_ff_s == 1
161              e_p = e_ff_p;
162           elseif e_ff_s == 2
163              if(sum(e_ff_p) == 5)
164                e_p = e_ff_p(floor((e_f_p-1)/2)+1);
165              else
166                inh = [1 2 4 3];
167                e_p = inh(e_ff_p(mod(floor((e_f_p)/2),2)+1));
168              end
169           end
170           xf_fine(k,1) = abs(x_fine(k) - old_x_fine(old_F_fine(e_ff,e_p)));
171         end
172
173         kap3 = b_fine'*(xf_fine);
174       end
175
176
177       %\tilde \mu 2 = ( ||\Pi h|| - ||h||)
178       tmu2 = hmin.* b.* (sum((x_fine(f2s)').^2)'...
179         -sum(repmat(sum(x_fine(f2s)',1)/4,4,1).^2)') /4;
180
```

```
181       % |||h/2 -h|||
182   %     eta = xd_fine'*A_fine*xd_fine;
183       eta = abs(xe_fine-xe);
184
185       end_time = toc - start_time;
186   %     disp(['Relative Zeit ' num2str(100*(build_time - start_time)/end_time)...
187   %       '% absolute Zeit '  t2str(build_time - start_time)]);
188       time_build = build_time - start_time;
189       time_solve = solve_time - build_time;
190
191       dataS = [dataS ...
192           typ(i) ... berechnet mit Typ
193           sqrt(sum(tmu))... tilde mu
194           sqrt(eta) ... eta
195           xe ... error (kappa 2)
196           sqrt(sum(mu))... mu
197           min(hmin)/max(hmax)...
198           min(hmax)/max(hmax)...
199           min(hmin./hmax)...
200           con... Kondition
201           sqrt(sum(tmu2))... tilde mu 2
202           xe_fine... (kappa)
203           kap3 ... kappa 3
204           time_build ... benoetigte Zeit (Aufbauen)
205           time_solve ... benoetigte Zeit (Berechnen)
206           ];
207     end
208
209     % markieren mit gewaehlten Parametern
210     marked = mark(x_fine(f2s)',tmu,theta,nu);
211
212     % Netz bunt plotten!
213   %   figure(1)
214   %   plotShape(G_C,G_E,'s',tmu);
215   %   title('Elemente mit Fehlerschaetzer')
216   %   colorbar
217   %   view(2)
218   %   plotMark(find(marked>1),G_C,G_E);
219
220     assert(size(elements,1)==length(marked),...
221       'MarkierungsVektor ist fehlerhaft')
222
223     old_C = coordinates;
224     old_E = elements;
225     old_S = sites;
226
227     %Netz verfeinern, wie durch marked bestimmt
228     [coordinates, elements, neigh, f, sites, er]...
229       = refineQuad(coordinates,elements,neigh,sites,marked);
230
231     %Vater Sohn test
232     assert(sum(2.^-sum(old_S,2))==sum(2.^-sum(sites,2)),...
233       'Gesamtinhalt Fehlerhaft')
234
235     e = 0;
236     for k = 1:size(data,1)
237       e = e +  abs(2.^-sum(old_S(k,:))...
238         - sum(2.^-sum(sites(unique(f(k,:)'),:),2)));
239     end
240     assert(e==0, 'Einzelne Inhalte Fehlerhaft')
```

```
241
242     % Fehlerhafte Elemente Plotten
243 %   if(~isempty(er))
244 %     figure(10)
245 %     plotShape(G_C,G_E,'');
246 %     plotMark(er,G_C,G_E);
247 %   end
248
249     p1 = find(file=='/',1,'last')+1;
250     if(isempty(p1))
251       p1=1;
252     end
253
254     p2 = find(file(p1:end)=='_',2)+p1;
255     if(length(p2)<2)
256        p2(2) = length(file);
257     else
258       p2(2) = p2(2) -2;
259     end
260
261   %ErgebnisWerte speichern
262   data(size(data,1)+1,1:length(dataS)) = dataS;
263   typeN = int2str(typ);
264   fileo = [typeN(typeN~=' ')...
265     't' regexprep(num2str(theta,2),'\.','')...
266     'n' regexprep(num2str(nu,2),'\.','') '_'...
267     file(p2(1):p2(2)) '_'];
268   save (['meshSave/' fileo int2str(size(data,1))]...
269     , 'coordinates', 'elements','neigh','data', 'sites')
270
271 end
272 end
273
274
275 function str = t2str(time)
276   type = 's';
277
278   if(time/60>1)
279     time =  time/60;
280     type = 'm';
281
282     if(time/60>1)
283       time =  time/60;
284       type = 'h';
285     end
286   end
287
288 str = [num2str(round(time*1000)/1000) type];
289 end
```

A.2.2 refineQuad.m

Diese Funktion implementiert den Verfeinern-Algorithmus 2.8. Übergeben werden vom Netz $\mathcal{T}_\ell$, die Koordinatenmatrix `coordinates` $\in \mathbb{R}^{M\times 3}$, die Elementmatrix `elements` $\in \{1,\dots,M\}^{N\times 4}$, die Nachbarschaftsrelationen `neigh` $\in \{0,1,\dots,N\}^{N\times 8}$. Weiterhin wird auch die Seitenmatrix `sites` $\in \mathbb{N}^{N\times 2}$ übergeben. Die $j$te Zeile (a_j, b_j) entspricht der Anzahl der Halbierungen der Seitenlängen a, b vom Element T_j, gelten muss hierbei $a = 2^{-a_j}$ und $b = 2^{-b_j}$. Außerdem entspricht `typ` $\in \{1,\dots,5\}^N$ der Markierung. Bei der Rückgabe des verfeinerten Netzes $\tilde{\mathcal{T}}_\ell$

werden zusätzlich die VaterSohn-Beziehungen `f2s` zurückgegeben.

```
function [coo,ele,nei,f2s,sit,err] =
    refineQuad(coordinates,elements,neigh,sites,typ)

err = [];

%Type: wenn nur ein Wert: aufblaehen
if([1 1] == size(typ))
    typ = repmat(typ, size(elements,1),1);
end

assert(size(elements,1)==size(neigh,1)&&size(elements,1)==length(typ),...
  'Dimensionen passen nicht');


%Globale Variabelen aufbauen
global G_ref_E;
global G_ref_C;
global G_ref_N;
global G_ref_S;
global G_ref_f2s; %finale Beziehung (VaterSohn)
global G_ref_t;  %wie soll verfeinert werden
global G_ref_tD; %wie wurde bereits verfeinert (in diesem Durchlauf)
global G_ref_f2sT; %temporaere Beziehung

%INTERNE globale Variablen zuweisen
G_ref_E = elements;
G_ref_C = coordinates;
G_ref_N = neigh;
G_ref_S = sites;
G_ref_t = typ;
G_ref_f2s = repmat((1:size(elements,1))',1,4);
G_ref_tD = ones(size(elements,1),1);

%Parameter freigeben (Speicher...)
clear elements coordinates neigh typ

% figure(11)
% plotShape(G_ref_C,G_ref_E,'s',G_ref_t);
% view(2)
% colorbar
% title('zuVerfeinern')

ref_old = [];
ref_old2 = [];

%jedes Element verfeinern
while(1==1)
  %jeden vierten 2er durch schrittweise Verfeinerung (5) ersetzen
  t_ref=find(G_ref_t==2);
  G_ref_t(t_ref(2:4:end)) = 5;

  %welche Elemente muessen bearbeitet werden
  ref = find(G_ref_t>1);
  ref = reshape(ref,1,length(ref));

  %muss noch weiter Verfeinert werden?
  if(isequal(ref,ref_old))
      assert(~isequal(ref_old2,G_ref_t,'Markierte sind verschieden'));
    break;
```

```
74    end
75    ref_old = ref;
76    ref_old2  = G_ref_t;
77    if(isempty(ref))
78      break;
79    end
80
81    % Elementeweise bearbeiten
82    for ele = ref % ref(randperm(length(ref)))
83
84      % HangingNode Check
85      Nt = find(G_ref_N(ele,5:8)==0);
86      N = G_ref_N(ele,Nt);
87      N2t = find(N~=0);
88      N2 = N(N2t); %Nachbarn der Kanten mit nur einem Nachbar
89      N2tt = Nt(N2t);  %Kante mit Nachbar ^
90
91      % Nur Nachbarn verfeinern an Kanten die geteilt werden
92      if(G_ref_t(ele)==3 || G_ref_t(ele)==5)
93         N2 = N2(mod(N2tt,2)==0);
94      elseif(G_ref_t(ele)==4)
95         N2 = N2(mod(N2tt,2)==1);
96      end
97
98      %hat noch zu teilende Nachbarn?
99      if(~isempty(N2))
100       N3t = mod(find((G_ref_N(N2',:)==ele)')-1,4)+1; %Nachbarseiten
101       N4t = find(diag(G_ref_N(N2',(N3t + 4)'))~=0)';  %Nachbarn mit 2Nachbarn
102         if(~isempty(N4t))
103           for i = N4t  %Elemente die noch verfeinert werden muessen
104             assert(G_ref_tD(N2(i))~=2,'Element wurde schon voll verfeinert')
105
106
107             %Element ist schon markiert
108             if(G_ref_t(N2(i)) == 2 || G_ref_t(N2(i)) ==5)
109                 continue;
110             end
111
112             %Wie muss das Element verfeinert werden.
113             if(G_ref_tD(N2(i)) ~= 1)
114               assert(G_ref_tD(N2(i))~=mod(N3t(i),2)+3,...
115                   'Element wurde in der Richtung schon verfeinert')
116               G_ref_t(N2(i)) = mod(N3t(i),2)+3;
117             elseif(G_ref_t(N2(i)) == 1)
118               G_ref_t(N2(i))=mod(N3t(i),2)+3;
119             elseif(G_ref_t(N2(i)) == mod(N3t(i)+1,2)+3)
120               G_ref_t(N2(i)) = 2;
121             end
122
123           end
124
125           % Da Nachbarn noch verfeinert werden muessen, erst mal weiter
126           continue;
127         end
128     end
129
130     % Wenn alle Ueberpruefungen durchgelaufen sind
131      assert(G_ref_tD(ele)~=2,'Element ist schon verfeinert')
132      assert(G_ref_t(ele)>1,'Element ist nicht Markiert')
133      G_ref_f2sT = ones(1,4)*ele;
```

```
      refineE(ele); %Element teilen
      updateN(ele); %Nachbarn des Elements aktualisieren
      updateF2S(ele); %VaterSohn Relation setzen
  end
end

%Rueckgabe zuweisen
coo = G_ref_C;
ele = G_ref_E;
nei = G_ref_N;
f2s = G_ref_f2s;
sit = G_ref_S;

%doppelte Koordinaten loeschen
[coo , ~, pos] = unique(coo,'rows');
pos = pos';
ele = pos(ele);

 %INTERNE globale Variablen freigeben
clear G_ref_E G_ref_C G_ref_N G_ref_f2s G_ref_t G_ref_tD G_ref_s
end

%% Element verfeinern ! sollte nur ausgefuehrt werden wenn wirklich moeglich
function refineE(ele)
% Element wird gnadenlos Verfeinert

global G_ref_E;
global G_ref_C;
global G_ref_S;
global G_ref_f2sT;
global G_ref_t;

    c_ele = size(G_ref_E,1);
    c_coo = size(G_ref_C,1);


    el = G_ref_E(ele,:);
    if(G_ref_t(ele)==2)
        G_ref_C(c_coo+1,:) = (G_ref_C(el(1),:)+G_ref_C(el(2),:))/2;
        G_ref_C(c_coo+2,:) = (G_ref_C(el(2),:)+G_ref_C(el(3),:))/2;
        G_ref_C(c_coo+3,:) = (G_ref_C(el(3),:)+G_ref_C(el(4),:))/2;
        G_ref_C(c_coo+4,:) = (G_ref_C(el(1),:)+G_ref_C(el(4),:))/2;
        G_ref_C(c_coo+5,:) = (G_ref_C(el(1),:)+G_ref_C(el(3),:))/2;

        G_ref_E(ele,:) = [c_coo+4,c_coo+5,c_coo+3,el(4)];
        G_ref_E(c_ele+1,:) = [c_coo+5,c_coo+2,el(3),c_coo+3];
        G_ref_E(c_ele+2,:) = [c_coo+1,el(2),c_coo+2,c_coo+5];
        G_ref_E(c_ele+3,:) = [el(1),c_coo+1,c_coo+5,c_coo+4];

        G_ref_S([ele (c_ele+1):(c_ele+3)]',:) = repmat(G_ref_S(ele,:)+[1 1],4,1);

        G_ref_f2sT(1:3)=c_ele+3:-1:c_ele+1;
    elseif(G_ref_t(ele)==3||G_ref_t(ele)==5)
        G_ref_C(c_coo+1,:) = (G_ref_C(el(1),:)+G_ref_C(el(4),:))/2;
        G_ref_C(c_coo+2,:) = (G_ref_C(el(2),:)+G_ref_C(el(3),:))/2;

        G_ref_E(ele,1) = c_coo+1;
        G_ref_E(ele,2) = c_coo+2;
        G_ref_E(c_ele+1,:) = [el(1),el(2),c_coo+2,c_coo+1];

```

```
194         G_ref_S([ele (c_ele+1)]',:) = repmat(G_ref_S(ele,:)+[0 1],2,1);
195
196         G_ref_f2sT([1 2])=c_ele+1;
197      elseif(G_ref_t(ele)==4)
198         G_ref_C(c_coo+1,:) = (G_ref_C(el(1),:)+G_ref_C(el(2),:))/2;
199         G_ref_C(c_coo+2,:) = (G_ref_C(el(4),:)+G_ref_C(el(3),:))/2;
200
201         G_ref_E(ele,2) = c_coo+1;
202         G_ref_E(ele,3) = c_coo+2;
203         G_ref_E(c_ele+1,:) = [c_coo+1,el(2),el(3),c_coo+2];
204
205         G_ref_S([ele (c_ele+1)]',:) = repmat(G_ref_S(ele,:)+[1 0],2,1);
206
207         G_ref_f2sT([2 3])=c_ele+1;
208     end
209 end
210
211
212 %% Aktualisieren der Nachbarn ! sollte nur ausgefuehrt werden wenn wirklich
        moeglich
213 function updateN(ele)
214 % Nachbarschaften werden neu gesetzt (nach N und f2s)
215
216 global G_ref_E;
217 global G_ref_N;
218 global G_ref_f2sT;
219 global G_ref_t;
220
221 this = G_ref_N(ele,:);
222 %an welchen Kanten habe ich Nachbarn
223 S = find(mod((this(1:4)~=0).*(this(5:8)==0),2))'; %einen Nachbar (Single)
224 D = find(this(5:8)~=0)';    %zwei Nachbarn (Double)
225
226 %an welchen Kanten bin ich Nachbar
227 MSt = mod(find((G_ref_N(this(S),:)==ele)')-1,8)+1;
228 MS = mod(MSt-1,4)+1;  % (Single)    %OHNE MOD????
229 MD = mod(find((G_ref_N(this([D D+4]),:)==ele)')-1,4)+1;
230 MD = reshape(MD,length(MD)/2,2); % (Double)
231
232 G_ref_N(G_ref_f2sT,1:8) = 0;
233
234 if(G_ref_t(ele)==3||G_ref_t(ele)==5)
235   G_ref_N(G_ref_f2sT([1 3])',1:4) = [ 0 0 G_ref_f2sT(3) 0;G_ref_f2sT(1) 0 0 0];
        %Innere Beziehung
236
237      % Beziehungen fuer Kanten mit einem Nachbar
238     for i = 1:length(S)
239       if(mod(S(i),2)==0)
240         G_ref_N(this(S(i)),[MS(i) MS(i)+4]) = [G_ref_f2sT(S(i))
              G_ref_f2sT(mod(S(i),4)+1)];
241         G_ref_N([G_ref_f2sT(S(i)) G_ref_f2sT(mod(S(i),4)+1)]',S(i))=this(S(i));
242       else
243         G_ref_N(this(S(i)),MSt(i)) = G_ref_f2sT(S(i));
244         G_ref_N(G_ref_f2sT(S(i)),S(i)) = this(S(i));
245       end
246     end
247
248     % Beziehungen fuer Kanten mit zwei Nachbarn
249     for i = 1:length(D)
250       if(mod(D(i),2)==0)
```

```
251         if(length(unique([G_ref_E(this(D(i)),:) G_ref_E(G_ref_f2sT(D(i)),:)]))==7)
252           G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(D(i));
253           G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(mod(D(i),4)+1);
254           G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i));
255           G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i)+4);
256         else
257           G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(mod(D(i),4)+1);
258           G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(D(i));
259           G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i)+4);
260           G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i));
261         end
262       else
263         G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(D(i));
264         G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(D(i));
265         G_ref_N(G_ref_f2sT(D(i)),[D(i) D(i)+4]) = [this(D(i)) this(D(i)+4)];
266       end
267     end
268
269 elseif(G_ref_t(ele)==4)
270   G_ref_N(G_ref_f2sT([1 2])',1:4) = [ 0 G_ref_f2sT(2) 0 0;0 0 0 G_ref_f2sT(1)];
271     % Beziehungen fuer Kanten mit einem Nachbar
272     for i = 1:length(S)
273       if(mod(S(i),2)==1)
274         G_ref_N(this(S(i)),[MS(i) MS(i)+4]) = [G_ref_f2sT(S(i))
              G_ref_f2sT(mod(S(i),4)+1)];
275         G_ref_N([G_ref_f2sT(S(i)) G_ref_f2sT(mod(S(i),4)+1)]',S(i))=this(S(i));
276       else
277         G_ref_N(this(S(i)),MSt(i)) = G_ref_f2sT(S(i));
278         G_ref_N(G_ref_f2sT(S(i)),S(i)) = this(S(i));
279       end
280     end
281
282     % Beziehungen fuer Kanten mit zwei Nachbarn
283     for i = 1:length(D)
284       if(mod(D(i),2)==1)
285         if(length(unique([G_ref_E(this(D(i)),:) G_ref_E(G_ref_f2sT(D(i)),:)]))==7)
286           G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(D(i));
287           G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(mod(D(i),4)+1);
288           G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i));
289           G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i)+4);
290         else
291           G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(mod(D(i),4)+1);
292           G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(D(i));
293           G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i)+4);
294           G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i));
295         end
296       else
297         G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(D(i));
298         G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(D(i));
299         G_ref_N(G_ref_f2sT(D(i)),[D(i) D(i)+4]) = [this(D(i)) this(D(i)+4)];
300       end
301     end
302
303 elseif(G_ref_t(ele)==2)
304   G_ref_N(G_ref_f2sT',1:4) = [0 G_ref_f2sT(2) G_ref_f2sT(4) 0; 0 0 G_ref_f2sT(3)
        G_ref_f2sT(1);...
305                             G_ref_f2sT(2) 0 0 G_ref_f2sT(4); G_ref_f2sT(1)
                                G_ref_f2sT(3) 0 0];
306
307     % Beziehungen fuer Kanten mit einem Nachbar
```

```
308     for i = 1:length(S)
309       G_ref_N(this(S(i)),[MS(i) MS(i)+4]) = [G_ref_f2sT(S(i))
              G_ref_f2sT(mod(S(i),4)+1)];
310       G_ref_N([G_ref_f2sT(S(i)) G_ref_f2sT(mod(S(i),4)+1)]',S(i))=this(S(i));
311     end
312
313     % Beziehungen fuer Kanten mit zwei Nachbarn
314     for i = 1:length(D)
315       if(length(unique([G_ref_E(this(D(i)),:) G_ref_E(G_ref_f2sT(D(i)),:)]))==7)
316         G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(D(i));
317         G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(mod(D(i),4)+1);
318         G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i));
319         G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i)+4);
320       else
321         G_ref_N(this(D(i)),MD(i,1)) = G_ref_f2sT(mod(D(i),4)+1);
322         G_ref_N(this(D(i)+4),MD(i,2)) = G_ref_f2sT(D(i));
323         G_ref_N(G_ref_f2sT(D(i)),D(i)) = this(D(i)+4);
324         G_ref_N(G_ref_f2sT(mod(D(i),4)+1),D(i)) = this(D(i));
325       end
326     end
327 end
328
329
330 end
331
332 %% aktualisieren der VaterSohn Beziehung ! sollte nur ausgefuehrt werden wenn
        wirklich moeglich
333 function updateF2S(ele)
334 %VaterSohn Beziehungen richtig Setzen
335     global G_ref_f2s;
336     global G_ref_f2sT;
337     global G_ref_t;
338     global G_ref_tD;
339
340     if(G_ref_tD(ele)==1) %wenn Element zum ersten Mal verfeinert wird
341         G_ref_f2s(ele,:) = G_ref_f2sT;
342         if(G_ref_t(ele)<5)
343             G_ref_tD(G_ref_f2sT) = G_ref_t(ele);
344             G_ref_t(G_ref_f2sT) = 0;
345         else %Element muss noch mal geteilt werden
346             G_ref_tD(G_ref_f2sT) = 3;
347             G_ref_t(G_ref_f2sT) = 4;
348         end
349     else %wenn Element zum zweiten Mal verfeinert wird
350         G_ref_tD(G_ref_f2sT) = 2;
351         org=floor((find(G_ref_f2s'==ele,1)-1)/4)+1;
352         pos=find(G_ref_f2s(org,:)==ele,1);
353
354         if(G_ref_t(ele)==3)
355             if(pos==1)
356                 G_ref_f2s(org,[1 4]) = G_ref_f2sT([1 3]);
357             else
358                 G_ref_f2s(org,[2 3]) = G_ref_f2sT([3 1]);
359             end
360         else
361             if(pos==1)
362                 G_ref_f2s(org,[1 2]) = G_ref_f2sT([1 2]);
363             else
364                 G_ref_f2s(org,[3 4]) = G_ref_f2sT([2 1]);
365             end
```

```
366         end
367         G_ref_t(G_ref_f2sT) = 0;
368     end
369 end
```

A.2.3 mark.m

Diese Funktion implementiert die Definition 6.3. Die Funktion wird mit der Matrix $xF2S$, dem Fehlerschätzer $\tilde{\mu}$ und den beiden Parametern θ, ν aufgerufen. Hierbei seien die Einträge der Matrix $xF2S \in \mathbb{R}^{N\times 4}$ geben durch $(xF2S)_{ij} = \langle \hat{\phi}_\ell, \chi_{(F2S_{ij})} \rangle$, wobei $F2S \in \mathbb{N}^{N\times 4}$ die Matrix der VaterSohn-Relationen und $\hat{\phi}_\ell$ die Galerkin-Lösung zum $\hat{\mathcal{T}}_\ell$ Netz ist. Zurückgeben wird der Markierungsvektor $marked \in \{1,2,3,4\}^N$.

```
1  function REF = mark(xF2S,ind,theta,nu)

9  if(size(xF2S,1)==1)
10   xF2S = xF2S';
11 end
12
13 T4 = [1 1 1 1; 1 -1 1 -1; 1 1 -1 -1;1 -1 -1 1]/4;
14
15 REF=ones(1,size(xF2S,2));
16 t1=zeros(1,size(xF2S,2));
17 t3 =0; t4 = 0;
18
19 Ct = T4*xF2S;
20
21 %% muss ueberhaupt verfeinert werden (welche sollen nicht verfeinert werden)
22 if(theta <1)
23     [s_ind idx] = sort(ind,'descend');
24
25     sum_ind = cumsum(s_ind,1);
26
27     ell = find(sum_ind  >= sum_ind(end) * theta,1);
28
29     %Symmetrisieren
30     ell = ell + find(abs(( sum_ind(ell)-sum_ind(ell:end)))/sum_ind(ell)>10^-2,1);
31
32     t1(idx(ell+1:end)) = 1;    % Nicht verfeinern
33 end
34
35
36 %% wie muss verfeinert werden
37 if(nu > 0) % Horizontal oder Vertikal
38     t3 = (nu*abs(Ct(3,:)) >= sqrt(Ct(2,:).^2+Ct(4,:).^2));
39     REF(t3-t1==1) = 3;
40     t4 = (nu*abs(Ct(4,:)) >= sqrt(Ct(2,:).^2+Ct(3,:).^2));
41     REF(t4-t1==1) = 4;
42 end
43 REF(~(t4+t3+t1)) = 2;    % Rest wird horizontal UND vertikal geteilt
44
45
46 end
```

Literatur

[1] Börm, S. und L. Grasedyck: *Low-Rank Approximation of Integral Operators by Interpolation*. Computing, 72:325–332, 2004.

[2] Dörfler, W.: *A Convergent Adaptive Algorithm for Poisson's Equation*. SIAM Journal on Numerical Analysis, 33(3):1106–1124, 1996.

[3] Ferraz-Leite, S. und D. Praetorius: *Simple a posteriori error estimators fot the h-version of the boundary element method*. Computing, 83(4):135–162, 2008.

[4] Maischak, M.: *The analytical computation of the Galerkin elements for the Laplace, Lamé and Helmholtz equation in 3D-BEM*. Techn. Ber., Institut für Angewandte Mthematik, University of Hannover, Juli 2000.

[5] Plato, R.: *Numerische Mathematik kompakt*. Vieweg Verlag, 2006.

[6] Praetorius, D.: *Hierarchische Matrizen und Fast Multipole Method*. 2009.

[7] Steinbach, O.: *Numerische Näherungsverfahren für elliptische Randwertprobleme*. Teubner & Springer Vieweg, 2003.