TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

BACHELORARBEIT

Stabile Implementierung von HILBERT S2P1

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

unter der Anleitung von
Ao.Univ.Prof. Dipl.Math. Dr. techn. Dirk Praetorius
Dr. techn. Thomas Führer
Dipl. Ing. Gregor Gantner

durch
Juliana Kainz
e1127254@student.tuwien.ac.at
Worathweg 22
4040 Linz

Inhaltsverzeichnis

1 Einleitung **4**

2 Berechnung bestimmter Doppelintegrale mittels Quadratur **5**
- 2.1 Interpolation 6
- 2.2 Gauß-Quadratur 7
- 2.3 Berechnung bestimmter Doppelintegrale mittels Quadratur 9
- 2.4 Approximierende Matrix 19
  - 2.4.1 Berechnung mit max-min-Kriterium 19
  - 2.4.2 Berechnung mit min-Kriterium 21

3 Lösungsverfahren und numerische Beispiele **22**
- 3.1 Problemstellung und Lösungsansatz 22
  - 3.1.1 Randelementmethode 22
  - 3.1.2 Galerkin-Verfahren 23
  - 3.1.3 Indirekte Randintegralmethode 28
  - 3.1.4 Fehlerschätzer 29
  - 3.1.5 Netzverfeinerung 30
  - 3.1.6 Lösungsalgorithmus 31
  - 3.1.7 Adaptiver Algorithmus 31
- 3.2 Numerische Beispiele 32
  - 3.2.1 Indirekte Randintegralmethode auf dem Schlitz 33
  - 3.2.2 Indirekte Randintegralmethode auf dem L-Shape 36
  - 3.2.3 Indirekte Randintegralmethode auf der Z-Shape 37

4 Vergleich der Implementierungen **42**
- 4.1 Implementierung des V-Operators 43
  - 4.1.1 `mexFunction` 44
  - 4.1.2 `computeVP1` 45
  - 4.1.3 `computeVij` 46
  - 4.1.4 `computeWij` 47
  - 4.1.5 `computeWijAnalytic` 47
  - 4.1.6 `computeWijSemianalytic` 48
  - 4.1.7 `computeWijFullQuadrature` 49
  - 4.1.8 `computeVP1P1ij` 50
  - 4.1.9 `computeVP1P1ijAnalytic` 51
  - 4.1.10 `computeVP1P1ijSemiAnalytic` 52
  - 4.1.11 `computeVP1P1ijFullQuadrature` 53
  - 4.1.12 `slp` 54
  - 4.1.13 `slpIterative` 54
  - 4.1.14 `doubleSlp` 56
- 4.2 Implementierung des K-Operators 58
  - 4.2.1 `mexFunction` 60
  - 4.2.2 `computeKP1S2` 61

4.2.3 computeKij 63
4.2.4 computeKijAnalytic 63
4.2.5 computeKijSwappedAnalytic 65
4.2.6 computeKijSemianalytic 66
4.2.7 computeKijSwappedSemianalytic 67
4.2.8 computeKijFullQuadrature 68
4.2.9 computeKP1S2ij 69
4.2.10 computeKP1S2ijAnalytic 70
4.2.11 computeKP1S2ijSemianalytic 71
4.2.12 computeKP1S2ijFullQuadrature 73
4.2.13 dlp 75
4.2.14 doubleDlp 76
4.2.15 doubleDlpSwapped 77

Literatur **80**

1 Einleitung

In dieser Arbeit beschäftigen wir uns mit der Randelementmethode zur Lösung der homogenen Laplace-Gleichung mit Dirichlet-Randbedingungen

$$\begin{aligned} -\Delta u &= 0 \qquad \text{in } \Omega \subset \mathbb{R}^2 \\ u &= g \qquad \text{auf } \Gamma := \partial\Omega, \end{aligned}$$

wobei $\Delta u := \partial_{x_1}^2 u + \partial_{x_2}^2 u$ den Laplace-Operator bezeichnet und $\Omega \subset \mathbb{R}^2$ eine beschränkte Teilmenge von $\mathbb{R}^2$ mit polygonalem Rand Γ ist.
Im Rahmen der Berechnung zerlegen wir den Rand in affine Randstücke $T_j = [\boldsymbol{a}, \boldsymbol{b}]$ wobei $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^2$ Punkte in der Ebene sind und $[\boldsymbol{a}, \boldsymbol{b}]$ die konvexe Hülle der Punkte $\boldsymbol{a}, \boldsymbol{b}$ bezeichnet. Für solche affinen Randstücke beschäftigen wir uns in Abschnitt 2 mit der Approximation von Integralen der Form

$$A_{jk}^{(\ell m)} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) \; \zeta_j^\ell(\boldsymbol{x}) \; \zeta_k^m(\boldsymbol{y}) \; ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \tag{1.1}$$

für Polynome $\zeta_j^\ell(\boldsymbol{x})$, $\zeta_k^m(\boldsymbol{y})$ vom Grad $\ell, m \in \mathbb{N}_0$ mit Träger auf den affinen Randstücken T_j, T_k. Für die Kernfunktion κ interessieren uns dabei Funktionen mit Singularitäten auf der Diagonalen, also für $\boldsymbol{x} = \boldsymbol{y}$. Im Speziellen werden wir die in Abschnitt 2 bewiesenen Resultate auf die Funktionen $\log|\boldsymbol{x} - \boldsymbol{y}|$ und $(\boldsymbol{y} - \boldsymbol{x})/\left|\boldsymbol{x} - \boldsymbol{y}\right|^2$ anwenden. Um das Doppelintegral aus (1.1) zu berechnen beschäftigen wir uns mit zwei unterschiedlichen Approximationsvarianten. Eine Möglichkeit ist, nur das äußere Integral über das kürzere der beiden Randstücke T_j, T_k durch Quadratur zu ersetzen. Alternativ können auch beide Integrale durch Quadratur ersetzt werden. Unter Voraussetzung bestimmter Zulässigkeitsbedingungen können wir zeigen, dass der angenäherte Wert bezüglich des gewählten Quadraturgrades exponentiell schnell gegen das exakte Integral konvergiert. Danach betrachten wir die Matrix $\mathbf{A}^{(\ell m)} \in \mathbb{R}^{n \times n}$ deren Einträge durch (1.1) festgelegt sind. Diese Matrix kann zum einen durch das *max-min-Kriterium* und zum anderen durch das *min-Kriterium* approximiert werden. Die Kriterien werden später genauer definiert. Im Prinzip sagen sie aus, dass in Abhängigkeit davon, welche Zulässigkeitsbedingungen gefordert werden, die Einträge $\mathbf{A}_{jk}^{(\ell m)}$ exakt oder approximativ berechnet werden. Dabei wird beim min-Kriterium maximal ein Integral durch Quadratur ersetzt, wohingegen beim max-min-Kriterium auch beide Integrale approximativ berechnet werden können. Für beide Kriterien zeigen wir, dass die approximativ berechnete Matrix $\mathbf{A}_p^{(\ell m)}$ bezüglich der Frobenius Norm für wachsenden Quadraturgrad exponentiell schnell gegen die exakte Matrix $\mathbf{A}^{(\ell m)}$ konvergiert.
In Abschnitt 3 erklären wir die Randelementmethode für die homogene Laplace-Gleichung mit Dirichlet-Randbedingungen. Hierfür gibt es zwei Möglichkeiten, die exakte Lösung zu berechnen. Eine Variante ist der direkte Ansatz über die Repräsentationsformel, die andere Möglichkeit ist die sogenannte indirekte Randintegralmethode. Um approximativ zu lösen, verwenden wir für beide Methoden das Galerkin-Verfahren. Im Zuge dieses numerischen Verfahrens müssen wir bestimmte Matrizen aufbauen, wozu wir die in Abschnitt 2 bewiesenen Resultate anwenden. Anschließend erklären wir noch einen Algorithmus, um ein vorgegebenes Startnetz adaptiv zu verfeinern und somit den Approximationsfehler zu optimieren. Anhand verschiedener numerischer Beispiele untersuchen wir dann die Verbesserungen, die eine Berechnung der benötigten Matrizen mittels max-min-Kriterium im Vergleich zum min-Kriterium bringt.
Zuletzt möchten wir noch in Abschnitt 4 auf die genaue Implementierung der benötigten Matrizen eingehen.

2 Berechnung bestimmter Doppelintegrale mittels Quadratur

Ziel dieses Abschnitts ist die Approximation des Integrals

$$A_{jk}^{(\ell m)} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \tag{2.1}$$

für Polynome $\zeta_j^\ell(\boldsymbol{x})$, $\zeta_k^m(\boldsymbol{y})$ vom Grad $\ell, m \in \mathbb{N}_0$ auf den affinen Randstücken T_j, T_k. Dabei werden jedoch gewisse Bedingungen an T_j, T_k und den Integranden $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ vorausgesetzt. Zur Approximation gibt es mehrere Möglichkeiten. Eine Variante ist, nur das äußere Integral durch Gauß-Quadratur zu ersetzten, im Gegensatz zur zweiten Variante, bei der beide Integrale durch Quadratur ersetzt werden. In diesem Abschnitt möchten wir zeigen, dass unter den erwähnten Voraussetzungen die exponentielle Konvergenz des Approximationsfehlers garantiert wird.

Bevor wir uns jedoch der Berechnung von konkreten Doppelintegralen mittels Quadratur widmen, möchten wir noch einige grundlegende Begriffe definieren, die wir in dieser Arbeit benötigen. Da im wesentlichen über zwei Strecken in der euklidischen Ebene integriert wird, möchten wir zunächst diesen Begriff genau definieren:

Definition 2.1. *Seien $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^2$ mit $\boldsymbol{a} \neq \boldsymbol{b}$. Dann heißt*

$$T := [\boldsymbol{a}, \boldsymbol{b}] := \left\{ \frac{1}{2}(\boldsymbol{a} + \boldsymbol{b} + \lambda(\boldsymbol{b} - \boldsymbol{a})) : \lambda \in [-1, 1] \right\}$$

affines Randstück *von $\boldsymbol{a}$ und $\boldsymbol{b}$.*

Im folgenden werden wir auch oft eine Parametrisierung eines affinen Randstücks brauchen. Die obige Definition legt für eine solche Parametrisierung schon eine Möglichkeit fest. Da wir im späteren Verlauf dieser Arbeit bestimmte Doppelintegrale betrachten, bei denen eine Funktion über zwei (meistens ungleiche) affine Randstücke integriert wird, möchten wir auch noch die gekoppelte Parametrisierung definieren.

Definition 2.2. *Seien $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{a}_k, \boldsymbol{b}_k \in \mathbb{R}^2$ mit $\boldsymbol{a}_j \neq \boldsymbol{b}_j$ und $\boldsymbol{a}_k \neq \boldsymbol{b}_k$. Seien weiters $T_j = [\boldsymbol{a}_j, \boldsymbol{b}_j]$, $T_k = [\boldsymbol{a}_k, \boldsymbol{b}_k]$ zwei affine Randstücke. Für das affine Randstück T_j sei die zugehörige Parametrisierung γ_j festgelegt durch*

$$\gamma_j : [-1, 1] \to T_j : \lambda \mapsto \frac{1}{2}(\boldsymbol{a}_j + \boldsymbol{b}_j + \lambda(\boldsymbol{b}_j - \boldsymbol{a}_j)).$$

Analog definiert man γ_k als Parametrisierung für T_k. Auf $T_j \times T_k$ sei die gekoppelte Parametrisierung *definiert durch*

$$\gamma_{jk} : [-1, 1]^2 \to T_j \times T_k : (x, y) \mapsto (\gamma_j(x), \gamma_k(y)).$$

Im folgenden verwenden wir manchmal den Abstand zwischen zwei Punkten im $\mathbb{R}^2$. Dieser Abstand ist durch die euklidische Norm gegeben, für welche wir nur $|\cdot|$ schreiben.
Für unsere Berechnung brauchen wir auch den Durchmesser bzw. die Länge eines affinen Randstücks und den Abstand zwischen zwei Randstücken.

Definition 2.3. *Seien $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{a}_k, \boldsymbol{b}_k \in \mathbb{R}^2$ mit $\boldsymbol{a}_j \neq \boldsymbol{b}_j$ und $\boldsymbol{a}_k \neq \boldsymbol{b}_k$. Seien weiters $T_j = [\boldsymbol{a}_j, \boldsymbol{b}_j]$, $T_k = [\boldsymbol{a}_k, \boldsymbol{b}_k]$ zwei affine Randstücke. Der* Durchmesser *bzw. die* Länge *von T_j sei gegeben durch*

$$\operatorname{diam}(T_j) = |\boldsymbol{b}_j - \boldsymbol{a}_j| \, .$$

Der Abstand *zwischen T_j und T_k sei definiert durch*

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

2.1 Interpolation

In diesem Abschnitt möchten wir die Chebyshev-Interpolation auf einem Intervall $[-1, 1]$ definieren.

Definition 2.4. *Seien $z_0, \dots, z_p \in [-1, 1]$ paarweise verschiedene* Knoten *und $y_0, \dots, y_p$ die zugehörigen* Funktionswerte*. Die* Langrange'sche Interpolationsaufgabe *lautet dann folgendermaßen: Finde ein Polynom q vom Grad p sodass*

$$q(z_i) = y_i \quad \text{für alle } i = 0, \dots, p.$$

Die Lagrange-Polynome *zu den Knoten $z_0, \dots, z_p \in [-1, 1]$ seien definiert durch*

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

In [4, Theorem 1.6] ist gezeigt, dass die Lagrange'sche Interpolationsaufgabe eine eindeutige Lösung hat, die durch

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

gegeben ist. Für eine Funktion $u : [-1, 1] \to \mathbb{R}$ können wir nun den Interpolationsoperator *$\mathcal{I}_p$ definieren:*

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

Außerdem sei die Lebesgue-Konstante *gegeben durch*

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

Bemerkung 2.5. *Wie man leicht sieht, ist der Interpolationsoperator ein linearer Operator.*

Da wir im späteren Verlauf dieser Arbeit immer wieder den Interpolationsfehler betrachten werden, möchten wir die Wahl der Knoten $z_0, \dots, z_p$ dahingehend optimieren, dass

der Interpolationsfeher minimal wird. Hierzu betrachten wir folgende einfache Fehlerabschätzung aus [4, Theorem 1.17] für $u \in \mathcal{C}^{p+1}([-1,1])$:

$$\|u - \mathcal{I}_p u\|_{L^\infty([-1,1])} \leq \max_{x\in[-1,1]} \left| \prod_{j=0}^{p} (x - z_j) \right| \frac{\|u^{(p+1)}\|_{L^\infty([-1,1])}}{(p+1)!}.$$

Diese Abschätzung wird laut [4, Theorem 1.24] möglichst scharf, wenn wir als Knoten genau die sogenannten *Chebyshev-Knoten* wählen. Diese sind definiert als die Nullstellen des $(p+1)$-ten Chebyshev-Polynoms und haben folgende Gestalt

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

Es gilt dann

$$\max_{x\in[-1,1]} \left| \prod_{j=0}^{p} (x - z_j) \right| = 2^{-p}.$$

Falls nicht anders definiert, meinen wir im folgenden mit $\mathcal{I}_p$ immer den Chebyshev-Interpolationsoperator.

Da wir später Funktionen in zwei Variablen betrachten, benötigen wir auch die komponentenweise Interpolation bzw. Interpolation nach zwei Variablen.

Definition 2.6. *Sei* $f : [-1,1]^2 \to \mathbb{R}, (x,y) \mapsto f(x,y)$ *eine stetige Funktion und* $(z_\ell)_{\ell=0}^p$ *Knoten in* $[-1,1]$ *. Dann sei die* komponentenweise Interpolation $\mathcal{I}_{p,x}$ *und* $\mathcal{I}_{p,y}$ *definiert durch*

$$(\mathcal{I}_{p,x} f)(x,y) := \sum_{j=0}^{p} f(z_j, y) L_j(x)$$
$$(\mathcal{I}_{p,y} f)(x,y) := \sum_{\ell=0}^{p} f(x, z_\ell) L_\ell(y).$$

Für die Interpolation nach beiden Variablen sei $\mathcal{I}_p$ *definiert durch:*

$$\begin{aligned}(\mathcal{I}_p f)(x,y) &:= \sum_{j=0}^{p} \sum_{\ell=0}^{p} f(z_j, z_\ell) L_j(x) L_\ell(y) \\ &= (\mathcal{I}_{p,x}\mathcal{I}_{p,y} f)(x,y) \\ &= (\mathcal{I}_{p,y}\mathcal{I}_{p,x} f)(x,y).\end{aligned}$$

Bemerkung 2.7. *Auch bei der komponentenweise Interpolation beziehungsweise der Interpolation nach mehreren Variablen wählen wir, wenn nicht anders definiert, Chebyshev-Knoten.*

2.2 Gauß-Quadratur

Für die Annäherung der Integrale aus (2.1) möchten wir die Gauß-Quadratur verwenden.

Definition 2.8. *Für Knoten $z_0, \dots, z_p \in [-1,1]$ und Gewichte $\omega_0, \dots, \omega_p \in \mathbb{R}$ versteht man unter einer* Quadratur *der Länge p die Approximation des Integrals*

$$\int_{-1}^{1} f(x)\omega(x)\; dx$$

durch

$$Q_p(f) := \sum_{k=0}^{p} \omega_k f(z_k).$$

Für diese Arbeit benötigen wir nur die klassische Gauß-Quadratur*, bei welcher zusätzlich $\omega \equiv 1$ vorausgesetzt ist und die Knoten z_k und Gewichte ω_k auf eine bestimmte Art gewählt werden (für Details siehe [4, Abschnitt 6.8]). Eine Quadratur sei* exakt *für eine Funktion f, falls $Q_p(f) = \int_{-1}^{1} f(x)\; dx$. Eine Quadratur hat* Exaktheitsgrad *m falls sie für alle Polynome vom Grad m exakt ist. Falls für alle $f \in \mathcal{C}([-1,1])$ gilt*

$$Q_p(f) = \int_{-1}^{1} q(x)\; dx,$$

wobei q das Interpolationspolynom von f vom Grad p zu den Knoten $z_0, \dots, z_p$ ist, so nennt man die Quadratur interpolatorisch.

Bemerkung 2.9. *Die Gauß-Quadratur der Länge $p \geq 0$ ist interpolatorisch und exakt vom Grad $2p+1$ (siehe [4, Korollar 6.38], wobei jedoch beachtet werden muss, dass dort in der Quadraturformel die Summation bei 1 beginnt anstatt bei 0).*

Wie bei der Interpolation möchten wir auch hier auf die zweidimensionale Quadratur eingehen.

Definition 2.10. *Sei $f : [-1,1]^2 \to \mathbb{R}, (x,y) \mapsto f(x,y)$ eine stetige Funktion. Zur Approximation von*

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\; dx\; dy$$

sei die komponetenweise Quadratur *$Q_{p,x}$ der Länge p definiert durch*

$$\begin{aligned} Q_{p,x}(f) &:= \int_{-1}^{1} \sum_{k=0}^{p} \omega_k f(z_k, y)\; dy \\ &= \sum_{k=0}^{p} \omega_k \int_{-1}^{1} f(z_k, y)\; dy. \end{aligned}$$

Analog definiert man $Q_{p,y}$. Zur approximativen Berechnung beider Integrale sei die zweidimensionale Quadratur *Q_p der* Länge *p definiert durch*

$$Q_p(f) := \sum_{k=0}^{p}\sum_{\ell=0}^{p} \omega_k \omega_\ell f(z_k, z_\ell).$$

Bemerkung 2.11. *Die zweidimensionale Quadratur ist im Prinzip nichts anderes als die Hintereinanderausführung von zwei Quadraturformeln. Das heißt, es gilt*

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\; dx\; dy \approx \int_{-1}^{1} \sum_{k=0}^{p} \omega_k f(z_k, y)\; dy \approx \sum_{k=0}^{p}\sum_{\ell=0}^{p} \omega_k \omega_\ell f(z_k, z_\ell).$$

2.3 Berechnung bestimmter Doppelintegrale mittels Quadratur

Bevor wir die exponentielle Konvergenz der mit Quadratur angenäherten Doppelintegrale zeigen können, benötigen wir noch einige Definitionen.

Definition 2.12. *Seien T_j, T_k affine Randstücke und sei $\eta > 0$. Dann heißt das Paar (T_j, T_k)* η-min-zulässig, *genau dann wenn*

$$\eta \operatorname{dist}(T_j, T_k) \geq \min\{\operatorname{diam}(T_j), \operatorname{diam}(T_k)\}. \tag{2.2}$$

Das Paar (T_j, T_k) heißt η-max-zulässig, *genau dann wenn*

$$\eta \operatorname{dist}(T_j, T_k) \geq \max\{\operatorname{diam}(T_j), \operatorname{diam}(T_k)\}. \tag{2.3}$$

Falls weder (2.2) *noch* (2.3) *gelten, so heißt das Paar (T_j, T_k)* η-unzulässig.

In folgender Bemerkung möchten wir kurz erläutern, warum wir die Doppelintegrale approximativ und nicht exakt berechnen.

Bemerkung 2.13. *Im Rahmen der Randelementmethode mit Galerkinverfahren ist es notwendig, die sogenannte Steifigkeitsmatrix zu berechnen. Die Einträge dieser Matrix für stückweise konstante Ansatzfunktionen sind durch*

$$A_{jk} := \int_{T_j} \int_{T_k} \log |\boldsymbol{x} - \boldsymbol{y}| \; ds_{\boldsymbol{y}} \, ds_{\boldsymbol{x}} \tag{2.4}$$

definiert. Anstatt die exakte Matrix zu berechnen, wird mittels Quadratur eine Annäherung A_p berechnet. Auf diese Art und Weise können Auslöschungseffekte vermieden werden, die vor allem für kurze Randstücke auftreten. Betrachten wir als Beispiel die „Stammfunktion“ des Integranden von (2.4) *für $T_k = [\boldsymbol{a}_k, \boldsymbol{b}_k]$*

$$F_{\boldsymbol{x}}(\boldsymbol{z}) := \int_{[\boldsymbol{a}_k, \boldsymbol{z}]} \log |\boldsymbol{x} - \boldsymbol{y}| \; ds_{\boldsymbol{y}}$$

für $\boldsymbol{x} \in \mathbb{R}^2$ fest. Offensichtlich ist die Stammfunktion für $\boldsymbol{z} \neq \boldsymbol{x}$ stetig. Als Folge davon gilt für T_k mit $\operatorname{diam}(T_k)$ klein allerdings $F_{\boldsymbol{x}}(\boldsymbol{a}_k) \approx F_{\boldsymbol{x}}(\boldsymbol{b}_k)$. Daher treten für die Berechnung

$$F_{\boldsymbol{x}}(\boldsymbol{b}_k) - F_{\boldsymbol{x}}(\boldsymbol{a}_k) = \int_{T_k} \log |\boldsymbol{x} - \boldsymbol{y}| \; ds_{\boldsymbol{y}}$$

starke Auslöschungseffekte auf, welche schlussendlich die Berechnung verfälschen würden. Praktisch können wir beobachten, dass eigentlich die Berechnung der Integrale mittels Gauß-Quadratur für η-min-zulässige bzw. η-max-zulässige Randstücke genauer ist, als die exakte Berechnung. Das liegt daran, dass bei der Gauß-Quadratur die Funktion $F_{\boldsymbol{x}}$ mehrmals ausgewertet und mit positiven Quadraturgewichten multipliziert wird. In der Praxis erreicht man laut [1, Bemerkung 3.14] durch Skalierung des Integrationsbereichs, dass die ausgewerteten Funktionswerte immer dasselbe Vorzeichen haben. Dadurch dass bei der Gauß-Quadratur also nur Werte mit gleichem Vorzeichen addiert werden, können Auslöschungseffekte vermieden werden.
Auch auf das Doppelintegral aus (2.4) *trifft diese Argumentation zu. Die Berechnung des Doppelintegrals lässt sich nach [10] auf mehrere Auswertungen des inneren Integrals zurückführen. Aus diesem Grund ist es aus numerischer Sicht am besten, zuerst über das längere der beiden Randstücke zu integrieren, was aufgrund der Vertauschbarkeit der Integrale erlaubt ist. So wird die Operation, die stärker auslöschungsbeahftet ist, an das Ende des Algorithmus gestellt.*

Bei der Betrachtung der Doppelintegrale sind besonders Integranden mit einer Singularität auf der Diagonalen interessant, die auf dem restlichen Definitionsbereich glatt sind. Nichtsdestotrotz möchten wir eine Abschätzung für die partiellen Ableitungen der Funktion erhalten. Hierzu ist folgende Definition nützlich.

Definition 2.14. *Für einen Multiindex* $\alpha \in \mathbb{N}_0^2$ *mit* $\alpha = (\alpha_1, \alpha_2)$ *sei* $|\alpha| := \alpha_1 + \alpha_2$ *und* $\alpha! := \alpha_1!\alpha_2!$.
Dann heißt die Kern-Funktion $\kappa(\boldsymbol{x}, \boldsymbol{y})$ asymptotisch glatt, *falls* $\kappa \in \mathcal{C}^\infty(\mathbb{R}^2 \times \mathbb{R}^2 \backslash \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^2 \times \mathbb{R}^2 : \boldsymbol{x} = \boldsymbol{y}\})$ *und Konstanten* $c_1, c_2 > 0$ *und eine Ordnung der Singularität* $s \geq 0$ *existieren, sodass*

$$\left|\partial_{\boldsymbol{x}}^\alpha \partial_{\boldsymbol{y}}^\beta \kappa(\boldsymbol{x}, \boldsymbol{y})\right| \leq c_1 \left(c_2 \left|\boldsymbol{x} - \boldsymbol{y}\right|\right)^{-(|\alpha|+|\beta|+s)} (\alpha + \beta)! \tag{2.5}$$

für alle Multiindizes $\alpha, \beta \in \mathbb{N}_0^2$ *mit* $|\alpha| + |\beta| \geq 1$ *gilt. Dabei sei*

$$\partial_{\boldsymbol{x}}^\alpha \partial_{\boldsymbol{y}}^\beta \kappa(\boldsymbol{x}, \boldsymbol{y}) := \partial_{x_1}^{\alpha_1} \partial_{x_2}^{\alpha_2} \partial_{y_1}^{\beta_1} \partial_{y_2}^{\beta_2} \kappa(\boldsymbol{x}, \boldsymbol{y}).$$

Ein wesentliches Argument für die exponentielle Konvergenz des Approximationsfehlers ist folgendes Lemma [2, Theorem 3.2], mit welchem wir später zeigen werden, dass asymptotisch glatte Kernfunktionen besonders gut durch Polynominterpolation angenähert werden können. Dabei sei $\mathcal{I}_p$ der d-dimensionale Chebyshev-Interpolationsoperator analog wie in Definition 2.6 erklärt.

Lemma 2.15 (Interpolationsfehler)**.** *Sei* $B \subseteq \mathbb{R}^d$ *ein achsenparalleler Quader mit*

$$B = I_1 \times \ldots \times I_d$$

wobei $(I_j)_{j=1}^d$ *kompakte Intervalle sind. Sei* $u \in C^\infty(B)$ *so, dass es Konstanten* $C_u, \gamma_u \geq 0$ *gibt mit*

$$\left\|\partial_j^n u\right\|_{L^\infty(B)} \leq C_u \gamma_u^n n! \tag{2.6}$$

für alle $j \in \{1, \ldots, d\}$ *und* $n \in \mathbb{N}_0$. *Dann erhalten wir für* $p \in \mathbb{N}$

$$\left\|u - \mathcal{I}_p u\right\|_{L^\infty(B)} \leq 8ed\Lambda_p^d C_u (1 + \gamma_u \mathrm{diam}(B))(p+1) \left(1 + \frac{2}{\gamma_u \mathrm{diam}(B)}\right)^{-(p+1)}.$$

Später möchten wir Lemma 2.15 auf eine Funktion $u := \kappa_{jk} := \kappa \circ \gamma_{jk}$ anwenden, wobei κ eine asymptotisch glatte Kernfunktion ist und γ_{jk} die gekoppelte Parametrisierung der affinen Randstücke T_j, T_k. Um die Voraussetzung (2.6) für κ_{jk} zu zeigen, müssen wir unter anderem die Ausdrücke $\partial_x^n \kappa_{jk}(x, y)$ und $\partial_y^n \kappa_{jk}(x, y)$ berechnen und danach geeignet abschätzen. In folgendem Lemma geben wir nicht nur die genaue Darstellung von $\partial_x^n \kappa_{jk}(x, y)$ und $\partial_y^n \kappa_{jk}(x, y)$ an, wir finden auch eine erste Abschätzung.

Lemma 2.16. *Seien* $\boldsymbol{a}_j, \boldsymbol{b}_j, \boldsymbol{a}_k, \boldsymbol{b}_k \in \mathbb{R}^2$ *mit* $\boldsymbol{a}_j \neq \boldsymbol{b}_j$ *und* $\boldsymbol{a}_k \neq \boldsymbol{b}_k$. *Seien weiters* $T_j = [\boldsymbol{a}_j, \boldsymbol{b}_j]$, $T_k = [\boldsymbol{a}_k, \boldsymbol{b}_k]$ *zwei affine Randstücke mit* $T_j \cap T_k = \emptyset$, *zugehöriger Parametrisierungen* γ_j, γ_k *und gekoppelter Parametrisierung* γ_{jk}. *Darüber hinaus sei* $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \kappa(\boldsymbol{x}, \boldsymbol{y})$ *eine asymptotisch glatte Kernfunktion mit Konstanten* c_1, c_2, s. *Bezeichne* $\kappa_{jk} := \kappa \circ \gamma_{jk}$. *Für* $r_1, \ldots, r_n \in \{x_1, x_2\}$ *sei die Anzahl, der* $r_k, k = 1, \ldots, n$ *welche* x_i *für*

$i = 1, 2$ annehmen, definiert durch $N_{x_i} := |\{r_k : r_k = x_i, k = 1, \dots, n\}|$. Analog sei N_{y_i} definiert. Dann gilt

$$\begin{aligned}\partial_x^n \kappa_{jk}(x,y) &= \sum_{r_1,\dots,r_n \in \{x_1,x_2\}} ((\partial_{r_1} \cdots \partial_{r_n} \kappa) \circ \gamma_{jk})(x,y) \left(\frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_1\right)^{N_{x_1}} \left(\frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_2\right)^{N_{x_2}} \\ &\leq c_1 (c_2 \, |\gamma_j(x) - \gamma_k(y)|)^{-(n+s)} n! \operatorname{diam}(T_j)^n\end{aligned} \tag{2.7}$$

beziehungsweise

$$\begin{aligned}\partial_y^n \kappa_{jk}(x,y) &= \sum_{r_1,\dots,r_n \in \{y_1,y_2\}} ((\partial_{r_1} \cdots \partial_{r_n} \kappa) \circ \gamma_{jk})(x,y) \left(\frac{1}{2}[\boldsymbol{b}_k - \boldsymbol{a}_k]_1\right)^{N_{y_1}} \left(\frac{1}{2}[\boldsymbol{b}_k - \boldsymbol{a}_k]_2\right)^{N_{y_2}} \\ &\leq c_1 (c_2 \, |\gamma_j(x) - \gamma_k(y)|)^{-(n+s)} n! \operatorname{diam}(T_k)^n.\end{aligned} \tag{2.8}$$

Dabei ist $[\cdot]_i$ der i-te Eintrag eines Vektors.

Beweis. Wir zeigen zuerst für $\partial_x^n \kappa_{jk}(x,y)$ die erste Gleichheit aus (2.7) mittels Induktion. Für $n = 1$ gilt also

$$\begin{aligned}\partial_x \kappa_{jk}(x,y) &= \partial_x (\kappa \circ \gamma_{jk})(x,y) \\ &= ((D\kappa) \circ \gamma_{jk})(x,y) \cdot (\partial_x \gamma_{jk})(x,y) \\ &= ((\partial_{x_1}\kappa, \partial_{x_2}\kappa, \partial_{y_1}\kappa, \partial_{y_2}\kappa) \circ \gamma_{jk})(x,y) \cdot \left(\frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_1, \frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_2, 0, 0\right)^T \\ &= ((\partial_{x_1}\kappa) \circ \gamma_{jk})(x,y) \frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_1 + ((\partial_{x_2}\kappa) \circ \gamma_{jk})(x,y) \frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_2.\end{aligned}$$

Somit ist der Induktionsanfang gezeigt. Die Induktionsvoraussetzung (IV) besagt dann, dass

$$\begin{aligned}&\partial_x^n \kappa_{jk}(x,y) \\ &= \sum_{r_1,\dots,r_n \in \{x_1,x_2\}} ((\partial_{r_1} \cdots \partial_{r_n} \kappa) \circ \gamma_{jk})(x,y) \left(\frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_1\right)^{N_{x_1}} \left(\frac{1}{2}[\boldsymbol{b}_j - \boldsymbol{a}_j]_2\right)^{N_{x_2}}\end{aligned} \tag{2.9}$$

bereits für n gilt. Wir zeigen nun (2.9) für $n + 1$. Mit der mehrdimensionalen Kettenregel

folgt

$$\begin{aligned}
&\partial_x^{n+1}\kappa_{jk}(x,y) = \partial_x(\partial_x\kappa_{jk})(x,y)\\
\overset{(IV)}{=}\ &\partial_x\left(\sum_{r_1,\dots,r_n\in\{x_1,x_2\}}((\partial_{r_1}\cdots\partial_{r_n}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}}\right)\\
=\ &\sum_{r_1,\dots,r_n\in\{x_1,x_2\}}\partial_x((\partial_{r_1}\cdots\partial_{r_n}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}}\\
=\ &\sum_{r_1,\dots,r_n\in\{x_1,x_2\}}((D(\partial_{r_1}\cdots\partial_{r_n}\kappa))\circ\gamma_{jk})(x,y)\cdot\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1,\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2,0,0\right)^T\\
&\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}}\\
=\ &\sum_{r_1,\dots,r_n\in\{x_1,x_2\}}((\partial_{x_1}\partial_{r_1}\cdots\partial_{r_n}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}+1}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}}+\\
&+((\partial_{x_2}\partial_{r_1}\cdots\partial_{r_n}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}+1}\\
=\ &\sum_{r_1,\dots,r_{n+1}\in\{x_1,x_2\}}((\partial_{r_1}\cdots\partial_{r_{n+1}}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_1\right)^{N_{x_1}}\left(\frac{1}{2}[\boldsymbol{b}_j-\boldsymbol{a}_j]_2\right)^{N_{x_2}}.
\end{aligned}$$

Damit haben wir nun (2.9) für $n+1$ gezeigt.
Es gilt $N_{x_1}+N_{x_2}=n$ und $|\{(r_k)_{k=1}^n : r_k\in\{x_1,x_2\}\}| = 2^n$. Da $[\boldsymbol{b}-\boldsymbol{a}]_1,[\boldsymbol{b}-\boldsymbol{a}]_2\le\operatorname{diam}(T_j)$ und κ eine asymptotisch glatte Kernfunktion mit Konstanten c_1,c_2,s ist, können wir diesen Ausdruck weiter abschätzen:

$$\begin{aligned}
\partial_x^n\kappa_{jk}(x,y) &\le \sum_{r_1,\dots,r_n\in\{x_1,x_2\}}((\partial_{r_1}\cdots\partial_{r_n}\kappa)\circ\gamma_{jk})(x,y)\left(\frac{\operatorname{diam}(T_j)}{2}\right)^n\\
&\le \sum_{r_1,\dots,r_n\in\{x_1,x_2\}}c_1(c_2\,|\gamma_j(x)-\gamma_k(y)|)^{-(n+s)}n!\left(\frac{\operatorname{diam}(T_j)}{2}\right)^n\\
&= c_1(c_2\,|\gamma_j(x)-\gamma_k(y)|)^{-(n+s)}n!\operatorname{diam}(T_j)^n.
\end{aligned}$$

Für $\partial_y^n\kappa_{jk}(x,y)$ lässt sich diese Argumentation analog durchführen und wir erhalten die gewünschte Abschätzung. □

Betrachten wir nun die semi-analytische Berechnung des Doppelintegrals, bei der nur das äußere Integral über das längere der beiden affinen Randstücke durch Quadratur ersetzt wird. Ähnlich wie in [1, Satz 3.7] möchten wir hier die exponentielle Konvergenz des Approximationsfehlers in Abhängigkeit des Quadraturgrades p zeigen.

Satz 2.17. *Seien T_j,T_k zwei η-min-zulässige affine Randstücke mit zugehöriger Parametrisierungen γ_j,γ_k, gekoppelter Parametrisierung γ_{jk} und* $\operatorname{diam}(T_j)\le\operatorname{diam}(T_k)$. *Sei weiters $\kappa:\mathbb{R}^2\times\mathbb{R}^2\to\mathbb{R}:(\boldsymbol{x},\boldsymbol{y})\mapsto\kappa(\boldsymbol{x},\boldsymbol{y})$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1,c_2,s. Darüber hinaus seien ζ_j^ℓ,ζ_k^m Polynome vom Grad $\ell,m\in\mathbb{N}_0$ mit Träger auf T_j,T_k.*

Weiters seien $\widetilde{\zeta}_j^\ell := \zeta_j^\ell \circ \gamma_j$, $\widetilde{\zeta}_k^m := \zeta_k^m \circ \gamma_k$ *und* $\kappa_{jk} := \kappa \circ \gamma_{jk}$. *Es sei* $A_{jk}^{(\ell m)}$ *durch*

$$\begin{aligned} A_{jk}^{(\ell m)} &= \int_{T_j}\int_{T_k} \kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}}\ ds_{\boldsymbol{x}} \\ &= \frac{\operatorname{diam}(T_j)}{2}\int_{-1}^{1}\int_{T_k} \kappa(\gamma_j(x),\boldsymbol{y})\ \widetilde{\zeta}_j^\ell(x)\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}}\ dx \end{aligned}$$

gegeben. Sei für $p \in \mathbb{N}_0$ *mit* $\ell - 1 \le p$ *der durch komponentenweise Gauß-Quadratur von* $A_{jk}^{(\ell m)}$ *zum Grad* p *entstehende Term*

$$(A_p)_{jk}^{(\ell m)} = \frac{\operatorname{diam}(T_j)}{2}\sum_{i=0}^{p}\omega_i\widetilde{\zeta}_j^\ell(z_i)\int_{T_k}\kappa(\gamma_j(z_i),\boldsymbol{y})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}}.$$

Dann gilt die Abschätzung

$$\begin{aligned} &\left|A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right| \\ &\le 16e\Lambda_p \left\|\zeta_k^m\right\|_{L^\infty(T_k)} \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \frac{c_1\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{(c_2\operatorname{dist}(T_j,T_k))^s}(p+1)\left(1+\frac{2\eta}{c_2}\right)\left(1+\frac{c_2}{\eta}\right)^{-(p+1)}. \end{aligned} \tag{2.10}$$

Beweis. Sei $\mathcal{I}_{p,x}$ der komponentenweise Chebyshev-Interpolationsoperator und Q_p die eindimensionale Gauß-Quadratur. Zusätzlich sei $\mathcal{J}_{p,x}$ der komponentenweise Lagrange-Interpolationsoperator, bei welchem die Stützstellen mit den Knoten $\{z_0,\dots,z_p\}$ der Gauß-Quadratur Q_p übereinstimmen. Wegen der Additivität des Integrals und der Tatsache, dass die Gauß-Quadratur Q_p interpolatorisch ist, folgt

$$\begin{aligned} &\left|A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right| \\ &= \frac{\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{4}\left|\int_{-1}^{1}\widetilde{\zeta}_k^m(y)\left[\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x)dx - Q_p\left(\kappa_{jk}(\cdot,y)\widetilde{\zeta}_j^\ell\right)\right]dy\right| \\ &= \frac{\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{4}\left|\int_{-1}^{1}\widetilde{\zeta}_k^m(y)\left[\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x) - \mathcal{J}_{p,x}\left(\kappa_{jk}\widetilde{\zeta}_j^\ell\right)(x,y)dx\right]dy\right| \\ &\le \frac{\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{4}\left\|\widetilde{\zeta}_k^m\right\|_{L^1([-1,1])}\sup_{y\in[-1,1]}\left|\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x) - \mathcal{J}_{p,x}\left(\kappa_{jk}\widetilde{\zeta}_j^\ell\right)(x,y)dx\right| \\ &\le \frac{\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{2}\left\|\zeta_k^m\right\|_{L^\infty(T_k)}\sup_{y\in[-1,1]}\left|\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x) - \mathcal{J}_{p,x}\left(\kappa_{jk}\widetilde{\zeta}_j^\ell\right)(x,y)dx\right|. \end{aligned}$$

Betrachten wir nun den Ausdruck über welchen das Supremum gebildet wird, genauer. Sei dafür $y \in [-1,1]$ beliebig aber fest. Dann erhalten wir

$$\begin{aligned} &\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x) - \mathcal{J}_{p,x}\left(\kappa_{jk}\widetilde{\zeta}_j^\ell\right)(x,y)dx \\ &= \underbrace{\int_{-1}^{1}\widetilde{\zeta}_j^\ell(x)\left[\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx}_{=:(i)} + \underbrace{\int_{-1}^{1}\widetilde{\zeta}_j^\ell(x)\mathcal{I}_{p,x}\kappa_{jk}(x,y)dx}_{=:(ii)} \\ &\quad - \underbrace{\int_{-1}^{1}\mathcal{J}_{p,x}\left[\widetilde{\zeta}_j^\ell\mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx}_{=:(iii)} - \underbrace{\int_{-1}^{1}\mathcal{J}_{p,x}\left[\widetilde{\zeta}_j^\ell\kappa_{jk} - \widetilde{\zeta}_j^\ell\mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx}_{=:(iv)} \end{aligned}$$

Diese vier Integrale können wir nun einzeln betrachten. Für (i) gilt mit der Hölder-Ungleichung:

$$\begin{aligned}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x) \left[\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx &\leq \left\|\widetilde{\zeta}_j^\ell\right\|_{L^1([-1,1])} \|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])} \\ &\leq 2\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])} \|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])}.\end{aligned}$$

Als nächstes betrachten wir (ii). Mit der Tatsache, dass $\widetilde{\zeta}_j^\ell(x)\mathcal{I}_{p,x}\kappa_{jk}(x,y)$ ein Polynom in x vom Grad $\ell + p \leq 2p+1$ ist und die Gauß-Quadratur exakt bis zu diesen Grad ist, folgt

$$\begin{aligned}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x)\mathcal{I}_{p,x}\kappa_{jk}(x,y)dx &= Q_p\left(\widetilde{\zeta}_j^\ell\mathcal{I}_{p,x}\kappa_{jk}(\cdot,y)\right) \\ &= \int_{-1}^{1} \mathcal{J}_{p,x}\left[\widetilde{\zeta}_j^\ell\mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx.\end{aligned}$$

Dabei haben wir für den letzten Schritt verwendet, dass Q_p interpolatorisch ist. Die rechte Seite stimmt nun mit (iii) überein und daher hebt sich $(ii) - (iii)$ auf.
Betrachten wir nun (iv). Mit der Tatsache, dass die Summe der Gewichte der Gauß-Qaudratur mit der Intervallbreite übereinstimmt, folgt

$$\begin{aligned}\int_{-1}^{1} \mathcal{J}_{p,x}\left[\widetilde{\zeta}_j^\ell\kappa_{jk} - \widetilde{\zeta}_j^\ell\mathcal{I}_{p,x}\kappa_{jk}\right](x,y)dx &= \sum_{i=0}^{p} \omega_i\widetilde{\zeta}_j^\ell(z_i)\left[\kappa_{jk}(z_i,y) - \mathcal{I}_{p,x}\kappa_{jk}(z_i,y)\right] \\ &\leq 2\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])} \|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])}.\end{aligned}$$

Insgesamt erhalten wir also

$$\begin{aligned}&\left|\int_{-1}^{1} \kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x) - \mathcal{J}_{p,x}\left(\kappa_{jk}(\cdot,y)\widetilde{\zeta}_j^\ell\right)(x)dx\right| \\ &\qquad\leq 4\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])} \|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])}.\end{aligned}$$

Dann können wir die Supremumsnorm $\|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])}$ mit Hilfe von Lemma 2.15 abschätzen. Wir wählen $B := [-1,1]$ mit $\operatorname{diam}(B) = 2$ und $u(x) := \kappa_{jk}(x,y)$. Mit Hilfe der Abschätzung (2.7) aus Lemma 2.16 und unter Verwendung von $|\gamma_j(x) - \gamma_k(y)| \geq \operatorname{dist}(T_j,T_k)$ folgt nun mit $\operatorname{diam}(T_j) = \min\{\operatorname{diam}(T_j), \operatorname{diam}(T_k)\}$ und der η-min-Zulässigkeit (2.2)

$$\begin{aligned}\left|u^{(n)}(x)\right| &= |\partial_x^n\kappa_{jk}(x,y)| \\ &\leq c_1(c_2|\gamma_j(x) - \gamma_k(y)|)^{-(n+s)}n!\operatorname{diam}(T_j)^n \\ &= c_1(c_2|\gamma_j(x) - \gamma_k(y)|)^{-s}n!\left(\frac{\operatorname{diam}(T_j)}{c_2|\gamma_j(x) - \gamma_k(y)|}\right)^n \\ &\leq \frac{c_1 n!}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(\frac{\operatorname{diam}(T_j)}{c_2\operatorname{dist}(T_j,T_k)}\right)^n \\ &\leq \frac{c_1 n!}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(\frac{\eta}{c_2}\right)^n.\end{aligned} \tag{2.11}$$

Mit den Konstanten

$$C_u := \frac{c_1}{(c_2\operatorname{dist}(T_j,T_k))^s} \quad \text{und} \quad \gamma_u := \frac{\eta}{c_2} \tag{2.12}$$

haben wir die Voraussetzung (2.6) für Lemma 2.15 gezeigt. Daher erhalten wir

$$\begin{aligned}\|u - \mathcal{I}_p u\|_{L^\infty([-1,1])} &\le 8e\Lambda_p C_u(1+2\gamma_u)(p+1)\left(1+\frac{1}{\gamma_u}\right)^{-(p+1)} \\ &= 8e\Lambda_p \frac{c_1}{(c_2 \mathrm{dist}(T_j,T_k))^s}\left(1+\frac{2\eta}{c_2}\right)(p+1)\left(1+\frac{c_2}{\eta}\right)^{-(p+1)}.\end{aligned}$$

Da $y \in [-1,1]$ beliebig war, gilt die obige Abschätzung auch für das Supremum über alle $y \in [-1,1]$. Insgesamt folgt daher

$$\begin{aligned}&\left|A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right| \\ &\le 2\mathrm{diam}(T_j)\mathrm{diam}(T_k)\,\|\zeta_k^m\|_{L^\infty(T_k)}\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\sup_{y\in[-1,1]}\|(\kappa_{jk} - \mathcal{I}_{p,x}\kappa_{jk})(\cdot,y)\|_{L^\infty([-1,1])} \\ &\le 16e\Lambda_p\,\|\zeta_k^m\|_{L^\infty(T_k)}\left\|\zeta_j^\ell\right\|_{L^\infty(T_j)}\frac{c_1\mathrm{diam}(T_j)\mathrm{diam}(T_k)}{(c_2\mathrm{dist}(T_j,T_k))^s}(p+1)\left(1+\frac{2\eta}{c_2}\right)\left(1+\frac{c_2}{\eta}\right)^{-(p+1)}\end{aligned}$$

und somit ist die Behauptung gezeigt. □

Für den Fall, dass nicht $\mathrm{diam}(T_j) \le \mathrm{diam}(T_k)$ gilt, sondern $\mathrm{diam}(T_k) < \mathrm{diam}(T_j)$, gehen wir wie folgt vor.
Laut nachfolgendem Lemma darf die Integrationsreihenfolge bei $A_{jk}^{\ell m}$ vertauscht werden. Weiters ist mit $(\boldsymbol{x},\boldsymbol{y}) \mapsto \kappa(\boldsymbol{x},\boldsymbol{y})$ auch $(\boldsymbol{x},\boldsymbol{y}) \mapsto \kappa(\boldsymbol{y},\boldsymbol{x})$ eine asymptotisch glatte Kernfunktion. Für die Abschätzung (2.10) muss $p \in \mathbb{N}_0$ allerdings so groß gewählt werden, dass $m-1 \le p$ gilt.

Lemma 2.18. *Seien T_j, T_k η-min-zulässige Randstücke und κ eine asymptotisch glatte Kernfunktion, die symmetrisch ist. Weiters seien ζ_j^ℓ, ζ_k^m Polynome vom Grad $\ell, m \in \mathbb{N}_0$ und*

$$A_{jk}^{(\ell m)} := \int_{T_j}\int_{T_k} \kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}.$$

Dann gilt

$$A_{jk}^{(\ell m)} = \int_{T_k}\int_{T_j} \kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{x}} ds_{\boldsymbol{y}}.$$

Beweis. Aus der η-min-Zulässigkeitsbedingung (2.2) folgt

$$\mathrm{dist}(T_j,T_k) \ge \frac{\min\{\mathrm{diam}(T_j),\mathrm{diam}(T_k)\}}{\eta} \ > \ 0$$

und daher insbesondere $T_j \cap T_k = \emptyset$. Da κ eine asymptotisch glatte Kernfunktion ist, ist sie glatt auf $T_j \times T_k$ und da ζ_j^ℓ und ζ_k^m Polynome sind, folgt, dass ihr Produkt wieder glatt ist. Die Menge $T_j \times T_k$ ist abgeschlossen und beschränkt und daher laut dem Satz von Heine-Borel kompakt. Laut [8, S. 249, Satz 6] ist $\kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})$ als stetige Abbildung auf einem Kompaktum integrierbar. Mit dem Satz von Fubini [8, S. 289] und der Symmetrie von κ folgt daher

$$\begin{aligned}A_{jk}^{(\ell m)} &= \int_{T_j}\int_{T_k} \kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &= \int_{T_k}\int_{T_j} \kappa(\boldsymbol{x},\boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{x}} ds_{\boldsymbol{y}},\end{aligned}$$

womit die Behauptung gezeigt ist. □

Im folgenden möchten wir nun zeigen, dass eine zu Satz 2.17 analoge Aussage auch gilt, falls wir beide Integrale mit Quadratur berechnen.

Satz 2.19. *Seien T_j, T_k zwei η-max-zulässige affine Randstücke mit zugehörigen Parametrisierungen γ_j, γ_k und gekoppelter Parametrisierung γ_{jk}. Sei weiters $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} : (\boldsymbol{x}, \boldsymbol{y}) \mapsto \kappa(\boldsymbol{x}, \boldsymbol{y})$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2 und $s \geq 0$. Darüber hinaus seien ζ_j^ℓ, ζ_k^m Polynome vom Grad $\ell, m \in \mathbb{N}_0$ auf T_j, T_k. Weiters seien $\widetilde{\zeta}_j^\ell := \zeta_j^\ell \circ \gamma_j$, $\widetilde{\zeta}_k^m := \zeta_k^m \circ \gamma_k$ und $\kappa_{jk} := (\kappa \circ \gamma_{jk})$. Es sei $A_{jk}^{(\ell m)}$ durch*

$$\begin{aligned} A_{jk}^{(\ell m)} &= \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y}) \; \zeta_j^\ell(\boldsymbol{x}) \; \zeta_k^m(\boldsymbol{y}) \; ds_{\boldsymbol{y}} \; ds_{\boldsymbol{x}} \\ &= \frac{\operatorname{diam}(T_j)}{2} \frac{\operatorname{diam}(T_k)}{2} \int_{-1}^{1} \int_{-1}^{1} \kappa_{jk}(x, y) \; \widetilde{\zeta}_j^\ell(x) \widetilde{\zeta}_k^m(y) \; dy \; dx \end{aligned}$$

gegeben. Sei für $p \in \mathbb{N}_0$ mit $\ell + m \leq p + 1$ der durch zweidimensionale Gauß-Quadratur von $A_{jk}^{(\ell m)}$ zum Grad p entstehende Term durch

$$(A_p)_{jk}^{(\ell m)} = \frac{\operatorname{diam}(T_j)}{2} \frac{\operatorname{diam}(T_k)}{2} \sum_{i=0}^{p} \sum_{h=0}^{p} \omega_i \omega_h \widetilde{\zeta}_j^\ell(z_i) \widetilde{\zeta}_k^m(z_h) \kappa_{jk}(z_i, z_h)$$

gegeben. Mit

$$C_{\eta jk} := \frac{c_1 \operatorname{diam}(T_j) \operatorname{diam}(T_k)}{(c_2 \operatorname{dist}(T_j, T_k))^s} \left(1 + \frac{2\sqrt{2}\eta}{c_2}\right) \tag{2.13}$$

gilt dann

$$\left| A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)} \right| \leq 32 e \Lambda_p^2 \left\| \zeta_j^\ell \right\|_{L^\infty(T_j)} \left\| \zeta_k^m \right\|_{L^\infty(T_k)} C_{\eta jk} (p+1) \left(1 + \frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}.$$

Beweis. Sei $\mathcal{I}_p$ der zweidimensionale Chebyshev-Interpolationsoperator und Q_p die zweidimensionale Gauß-Quadratur. Zusätzlich sei $\mathcal{J}_p$ der zweidimensionale Lagrange-Interpolationsoperator bei welchem die Stützstellen mit den Knoten $\{z_0, \ldots, z_p\}$ der Gauß-Quadratur Q_p übereinstimmen. Da die Gauß-Quadratur interpolatorisch ist, gilt

$$\begin{aligned} &\left| A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)} \right| \\ &= \frac{\operatorname{diam}(T_j)}{2} \frac{\operatorname{diam}(T_k)}{2} \left| \int_{-1}^{1} \int_{-1}^{1} \kappa_{jk}(x, y) \widetilde{\zeta}_j^\ell(x) \widetilde{\zeta}_k^m(y) dy \; dx - Q_p \left[\kappa_{jk} \widetilde{\zeta}_j^\ell \widetilde{\zeta}_k^m \right] \right| \\ &= \frac{\operatorname{diam}(T_j)}{2} \frac{\operatorname{diam}(T_k)}{2} \left| \int_{-1}^{1} \int_{-1}^{1} \kappa_{jk}(x, y) \widetilde{\zeta}_j^\ell(x) \widetilde{\zeta}_k^m(y) - \mathcal{J}_p \left[\kappa_{jk} \widetilde{\zeta}_j^\ell \widetilde{\zeta}_k^m \right] (x, y) \; dy \; dx \right|. \end{aligned}$$

Durch Addieren und Subtrahieren geeigneter Terme erhalten wir

$$\begin{aligned}
&\int_{-1}^{1}\int_{-1}^{1} \kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y) - \mathcal{J}_p\left[\kappa_{jk}\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\right](x,y)\,dy\,dx \\
&= \underbrace{\int_{-1}^{1}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)\left[\kappa_{jk} - \mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx}_{=:(i)} \\
&\quad + \underbrace{\int_{-1}^{1}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)\mathcal{I}_p\kappa_{jk}(x,y)\,dy\,dx}_{=:(ii)} \\
&\quad - \underbrace{\int_{-1}^{1}\int_{-1}^{1} \mathcal{J}_p\left[\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx}_{=:(iii)} \\
&\quad - \underbrace{\int_{-1}^{1}\int_{-1}^{1} \mathcal{J}_p\left[\kappa_{jk}\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m - \widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx}_{=:(iv)}.
\end{aligned}$$

Nun können wir diese vier Doppelintegrale einzeln betrachten:
Für das Doppelintegral (i) gilt

$$\begin{aligned}
&\int_{-1}^{1}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)\left[\kappa_{jk} - \mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx \\
&\le \left\|\widetilde{\zeta}_j^\ell\right\|_{L^1([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^1([-1,1])}\left\|\kappa_{jk} - \mathcal{I}_p\kappa_{jk}\right\|_{L^\infty([-1,1]^2)} \\
&\le 4\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^\infty([-1,1])}\left\|\kappa_{jk} - \mathcal{I}_p\kappa_{jk}\right\|_{L^\infty([-1,1]^2)}.
\end{aligned}$$

Als nächstes betrachten wir das Doppelintegral (ii). Mit der Tatsache, dass das Polynom $\widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)\mathcal{I}_p\kappa_{jk}(x,y)$ vom Grad $\le 2p+1$ in x und in y ist, die Gauss-Quadratur exakt bis zu diesem Grad und interpolatorisch ist, erhalten wir

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)\mathcal{I}_p\kappa_{jk}(x,y)\,dy\,dx &= Q_p\left(\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\mathcal{I}_p\kappa_{jk}\right) \\
&= \int_{-1}^{1}\int_{-1}^{1} \mathcal{J}_p\left[\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx.
\end{aligned}$$

Der Ausdruck auf der rechten Seite stimmt nun mit dem Doppelintegral (iii) überein und daher hebt sich $(ii) - (iii)$ auf.
Für das Doppelintegral (iv) gilt mit der Tatsache, dass die Summe der Gauß-Gewichte mit der Intervallbreite übereinstimmt

$$\begin{aligned}
&\int_{-1}^{1}\int_{-1}^{1} \mathcal{J}_p\left[\kappa_{jk}\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m - \widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\mathcal{I}_p\kappa_{jk}\right](x,y)\,dy\,dx \\
&= \sum_{i=0}^{p}\sum_{h=0}^{p} \omega_i\omega_h\widetilde{\zeta}_j^\ell(z_i)\widetilde{\zeta}_k^m(z_h)\left[\kappa_{jk}(z_i,z_h) - \mathcal{I}_p\kappa_{jk}(z_i,z_h)\right] \\
&\le 4\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^\infty([-1,1])}\left\|\kappa_{jk} - \mathcal{I}_p\kappa_{jk}\right\|_{L^\infty([-1,1]^2)}.
\end{aligned}$$

Insgesamt folgt also

$$\begin{aligned}&\left|\int_{-1}^{1}\int_{-1}^{1}\kappa_{jk}(x,y)\widetilde{\zeta}_j^\ell(x)\widetilde{\zeta}_k^m(y)-\mathcal{J}_p\left[\kappa_{jk}\widetilde{\zeta}_j^\ell\widetilde{\zeta}_k^m\right](x,y)\;dy\;dx\right|\\&\le 8\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^\infty([-1,1])}\|\kappa_{jk}-\mathcal{I}_p\kappa_{jk}\|_{L^\infty([-1,1]^2)}.\end{aligned}$$

Die Supremumsnorm $\|\kappa_{jk}-\mathcal{I}_p\kappa_{jk}\|_{L^\infty([-1,1]^2)}$ können wir nun mit Hilfe von Lemma 2.15 weiter abschätzen. Dazu wählen wir $B:=[-1,1]^2$ mit $\operatorname{diam}(B)=2\sqrt{2}$ und $u:=\kappa_{jk}$. Falls wir also (2.6) mit Konstanten $C_u,\gamma_u\ge 0$ für u zeigen können, folgt:

$$\|\kappa_{jk}-\mathcal{I}_p\kappa_{jk}\|_{L^\infty([-1,1]^2)}\le 16e\Lambda_p^2C_u(1+\gamma_u2\sqrt{2})\;(p+1)\left(1+\frac{1}{\gamma_u\sqrt{2}}\right)^{-(p+1)}. \tag{2.14}$$

Mit Hilfe der Abschätzung (2.7) aus Lemma 2.16 erhalten wir nun analog zu (2.11) die folgende Abschätzung:

$$\begin{aligned}|\partial_x^n u(x,y)|&=|\partial_x^n\kappa_{jk}(x,y)|\\&\le\frac{c_1 n!}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(\frac{\eta}{c_2}\right)^n.\end{aligned}$$

Offensichtlich gilt auch für die Ableitung nach der zweiten Komponenten

$$\left|\partial_y^n u(x,y)\right|\le\frac{c_1 n!}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(\frac{\eta}{c_2}\right)^n.$$

Mit den Konstanten

$$C_u:=\frac{c_1}{(c_2\operatorname{dist}(T_j,T_k))^s}\quad\text{und}\quad\gamma_u:=\frac{\eta}{c_2} \tag{2.15}$$

haben wir also die Voraussetzung (2.6) für Lemma 2.15 gezeigt und daher gilt (2.14). In weiterer Folge erhalten wir

$$\begin{aligned}&\left|A_{jk}^{(\ell m)}-(A_p)_{jk}^{(\ell m)}\right|\\&\le 2\operatorname{diam}(T_j)\operatorname{diam}(T_k)\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^\infty([-1,1])}\|\kappa_{jk}-\mathcal{I}_p\kappa_{jk}\|_{L^\infty([-1,1]^2)}\\&\le 32e\Lambda_p^2\left\|\widetilde{\zeta}_j^\ell\right\|_{L^\infty([-1,1])}\left\|\widetilde{\zeta}_k^m\right\|_{L^\infty([-1,1])}\\&\quad\frac{c_1\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(1+\frac{2\sqrt{2}\eta}{c_2}\right)(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}\\&=32e\Lambda_p^2\left\|\zeta_j^\ell\right\|_{L^\infty(T_j)}\|\zeta_k^m\|_{L^\infty(T_k)}\\&\quad\frac{c_1\operatorname{diam}(T_j)\operatorname{diam}(T_k)}{(c_2\operatorname{dist}(T_j,T_k))^s}\left(1+\frac{2\sqrt{2}\eta}{c_2}\right)(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)},\end{aligned}$$

womit der Satz bewiesen ist. □

Bemerkung 2.20. *Laut [7, Theorem 1.2] gilt für die Chebyshev-Interpolation* $\Lambda_p\le 2\log(p+1)/\pi+1$*, das heißt* Λ_p *wächst logarithmisch in* p*. Daher konvergieren* $(A_p)_{jk}^{(\ell m)}$ *aus Satz 2.17 und beziehungsweise Satz 2.19 für wachsenden Quadraturgrad* p *exponentiell schnell gegen* $A_{jk}^{(\ell m)}$*.*

2.4 Approximierende Matrix

In diesem Abschnitt betrachten wir die Situation, dass $T_1, \dots, T_n$ affine Randstücke sind. Wir möchten eine bestimmte Matrix A durch einen approximierende Matrix A_p annähern und abschließend mit Hilfe der vorhin gezeigten Resultate zeigen, dass A_p exponentiell schnell bezüglich der Frobenius-Norm gegen A konvergiert.
Es gibt zwei Möglichkeiten, die approximierende Matrix zu definieren, einmal mit dem sogenannten max-min-Kriterium und einmal mit dem min-Kriterium.

2.4.1 Berechnung mit max-min-Kriterium

Definition 2.21. *Seien $T_1, T_2, \dots, T_n$ affine Randstücke, κ eine asymptotisch glatte Kernfunktion und ζ_j^ℓ, ζ_k^m Polynome vom Grad $\ell, m \in \mathbb{N}_0$. Weiters sei $A^{(\ell m)} \in \mathbb{R}^{n \times n}$ gegeben durch*

$$A_{jk}^{(\ell m)} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}. \tag{2.16}$$

Sei der Quadraturgrad der Gauß-Quadratur $p \in \mathbb{N}_0$ so groß, dass $\ell + m \le p + 1$, dann ist die $A^{(\ell m)}$ approximierende Matrix nach dem max-min-Kriterium *$A_p^{(\ell m)}$ folgendermaßen definiert:*

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

$$(A_p)_{jk}^{(\ell m)} := A_{jk}^{(\ell m)} \tag{2.17}$$

- *Sind T_j und T_k η-*min-zulässig*, aber nicht η-max-zulässig, und* $\operatorname{diam}(T_j) \le \operatorname{diam}(T_k)$, *so ist*

$$(A_p)_{jk}^{(\ell m)} := \frac{\operatorname{diam}(T_j)}{2} \sum_{i=0}^{p} \omega_i \widetilde{\zeta}_j^\ell(z_i) \int_{T_k} \kappa(\gamma_j(z_i), \boldsymbol{y})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} \tag{2.18}$$

- *Sind hingegen T_j und T_k η-*min-zulässig*, aber nicht η-max-zulässig, und* $\operatorname{diam}(T_j) > \operatorname{diam}(T_k)$, *so ist*

$$(A_p)_{jk}^{(\ell m)} := \frac{\operatorname{diam}(T_k)}{2} \sum_{i=0}^{p} \omega_i \widetilde{\zeta}_k^m(z_i) \int_{T_j} \kappa(\boldsymbol{x}, \gamma_k(z_i))\ \zeta_j^\ell(\boldsymbol{x})\ ds_{\boldsymbol{x}} \tag{2.19}$$

- *Sind T_j und T_k η-*max-zulässig*, so ist*

$$(A_p)_{jk}^{(\ell m)} := \frac{\operatorname{diam}(T_j)}{2} \frac{\operatorname{diam}(T_k)}{2} \sum_{i=0}^{p} \sum_{h=0}^{p} \omega_i \omega_h \widetilde{\zeta}_j^\ell(z_i) \widetilde{\zeta}_k^m(z_h) \kappa(\gamma_j(z_i), \gamma_k(z_h)) \tag{2.20}$$

Wir wollen nun zeigen, dass für wachsenden Quadraturgrad die approximierende Matrix bezüglich der Frobeniusnorm exponentiell schnell gegen die gegebene Matrix $A^{(\ell m)}$ konvergiert. Die Frobenius-Norm ist hierbei gegeben durch

$$\|A\|_{\mathcal{F}} := \left(\sum_{j=1}^{n} \sum_{k=1}^{n} A_{jk}^2 \right)^{1/2} \qquad \text{für } A \in \mathbb{R}^{n \times n}.$$

Satz 2.22. *Seien $T_1, T_2, \ldots, T_n$ affine Randstücke. Sei $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2 und Singularitätsordnung $s \geq 0$. Für $\ell, m \in \mathbb{N}_0$ sei $A^{(\ell m)} \in \mathbb{R}^{n\times n}$ eine Matrix, deren Einträge durch (2.16) gegeben sind. Sei für $p \in \mathbb{N}_0$ mit $\ell + m \leq p+1$, die approximierende Matrix $A_p^{(\ell m)}$ gemäß Definition 2.21 gegeben. Weiters sei $C_{\eta jk} > 0$ definiert durch (2.13). Dann gilt*

$$\begin{aligned}&\left\|A^{(\ell m)} - A_p^{(\ell m)}\right\|_{\mathcal{F}}\\ &\leq n \max_{j=1,\ldots,n} \max_{k=1,\ldots,n} 32e\Lambda_p^2 \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \left\|\zeta_k^m\right\|_{L^\infty(T_k)} C_{\eta jk}(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}.\end{aligned} \tag{2.21}$$

Beweis. Betrachten wir zunächst die Differenz zwischen $A^{(\ell m)}$ und $A_p^{(\ell m)}$ in einem festen Eintrag. Falls T_j und T_k η-unzulässig sind, so ist die Differenz laut Definition 0.
Falls T_j und T_k η-min-zulässig sind, aber nicht η-max-zulässig, so können wir zwei Fälle unterscheiden. Ist $\mathrm{diam}(T_j) \leq \mathrm{diam}(T_k)$, so ist $(A_p)_{jk}^{(\ell m)}$ durch (2.18) gegeben. Nach Satz 2.17 und da $\Lambda_p \geq 1$ gilt

$$\begin{aligned}&\left|A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right|\\ &\leq 16e\Lambda_p \left\|\zeta_k^m\right\|_{L^\infty(T_k)} \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)}\\ &\quad \frac{c_1 \mathrm{diam}(T_j)\mathrm{diam}(T_k)}{(c_2 \mathrm{dist}(T_j,T_k))^s}(p+1)\left(1+\frac{2\eta}{c_2}\right)\left(1+\frac{c_2}{\eta}\right)^{-(p+1)}\\ &\leq 32e\Lambda_p^2 \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \left\|\zeta_k^m\right\|_{L^\infty(T_k)} C_{\eta jk}(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}.\end{aligned} \tag{2.22}$$

Für den Fall, dass $\mathrm{diam}(T_j) > \mathrm{diam}(T_k)$, gilt wegen Lemma 2.18

$$A_{jk}^{(\ell m)} = \int_{T_k}\int_{T_j} \kappa(\boldsymbol{x},\boldsymbol{y})\zeta_k^m(\boldsymbol{y})\zeta_j^\ell(\boldsymbol{x})ds_{\boldsymbol{x}}ds_{\boldsymbol{y}}.$$

Der approximierende Eintrag $(A_p)_{jk}^{(\ell m)}$ ist durch (2.19) gegeben. Durch Anwendung von Satz 2.17 erhalten wir wieder dieselbe Abschätzung wie in (2.22).
Für den Fall, dass T_j und T_k η-max-zulässig sind, ist $(A_p)_{jk}^{(\ell m)}$ durch (2.20) gegeben. Laut Satz 2.19 gilt

$$\left|A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right| \leq 32e\Lambda_p^2 \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \left\|\zeta_k^m\right\|_{L^\infty(T_k)} C_{\eta jk}(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}. \tag{2.23}$$

Insgesamt erhalten wir

$$\begin{aligned}&\left\|A^{(\ell m)} - A_p^{(\ell m)}\right\|_{\mathcal{F}}^2\\ &= \sum_{j,k=1}^n \left(A_{jk}^{(\ell m)} - (A_p)_{jk}^{(\ell m)}\right)^2\\ &\leq \sum_{j,k=1}^n \left(32e\Lambda_p^2 \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \left\|\zeta_k^m\right\|_{L^\infty(T_k)} C_{\eta jk}(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}\right)^2\\ &\leq n^2 \max_{j=1,\ldots,n}\max_{k=1,\ldots,n} \left(32e\Lambda_p^2 \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \left\|\zeta_k^m\right\|_{L^\infty(T_k)} C_{\eta jk}(p+1)\left(1+\frac{c_2}{\sqrt{2}\eta}\right)^{-(p+1)}\right)^2.\end{aligned}$$

Indem wir nun auf beiden Seiten die Wurzel ziehen bekommen wir die gewünschte Abschätzung. □

2.4.2 Berechnung mit min-Kriterium

Definition 2.23. *Seien $T_1, T_2, \dots, T_n$ affine Randstücke, κ eine asymptotisch glatte Kernfunktion und ζ_j^ℓ, ζ_k^m Polynome vom Grad $\ell, m \in \mathbb{N}_0$. Weiters sei $A^{(\ell m)} \in \mathbb{R}^{n\times n}$ gegeben durch*

$$A_{jk}^{(\ell m)} = \int_{T_j} \int_{T_k} \kappa(\boldsymbol{x}, \boldsymbol{y})\ \zeta_j^\ell(\boldsymbol{x})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}. \tag{2.24}$$

Sei der Quadraturgrad $p \in \mathbb{N}_0$ so groß, dass $\ell + m \le p+1$, dann ist die $A^{(\ell m)}$ approximierende Matrix nach dem min-Kriterium $A_p^{(\ell m)}$ *folgendermaßen definiert:*

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

$$(A_p)_{jk}^{(\ell m)} := A_{jk}^{(\ell m)}. \tag{2.25}$$

- *Sind T_j und T_k η-*min-zulässig *und* $\operatorname{diam}(T_j) \le \operatorname{diam}(T_k)$*, so ist*

$$(A_p)_{jk}^{(\ell m)} := \frac{\operatorname{diam}(T_j)}{2} \sum_{i=0}^{p} \omega_i \widetilde{\zeta}_j^\ell(z_i) \int_{T_k} \kappa(\gamma_j(z_i), \boldsymbol{y})\ \zeta_k^m(\boldsymbol{y})\ ds_{\boldsymbol{y}}. \tag{2.26}$$

- *Sind hingegen T_j und T_k η-*min-zulässig *und* $\operatorname{diam}(T_j) > \operatorname{diam}(T_k)$*, so ist*

$$(A_p)_{jk}^{(\ell m)} := \frac{\operatorname{diam}(T_k)}{2} \sum_{i=0}^{p} \omega_i \widetilde{\zeta}_k^m(z_i) \int_{T_j} \kappa(\boldsymbol{x}, \gamma_k(z_i))\ \zeta_j^\ell(\boldsymbol{x})\ ds_{\boldsymbol{x}}. \tag{2.27}$$

Auch hier konvergiert die approximierende Matrix für wachsenden Quadraturgrad bezüglich der Frobeniusnorm exponentiell schnell gegen die gegebene Matrix $A^{(\ell m)}$.

Satz 2.24. *Seien $T_1, T_2, \dots, T_n$ affine Randstücke. Sei $\kappa : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ eine asymptotisch glatte Kernfunktion mit Konstanten c_1, c_2 und Singularitätsordnung $s \ge 0$. Für $\ell, m \in \mathbb{N}_0$ sei $A^{(\ell m)} \in \mathbb{R}^{n\times n}$ eine Matrix, deren Einträge durch (2.24) gegeben sind. Weiters sei für $p \in \mathbb{N}_0$ mit $\ell, m \le p+1$, die approximierende Matrix $A_p^{(\ell m)}$ gemäß Definition 2.21 erklärt. Dann gilt*

$$\begin{aligned} &\left\|A^{(\ell m)} - A_p^{(\ell m)}\right\|_{\mathcal{F}} \\ &\le 16e\Lambda_p \left\|\zeta_k^m\right\|_{L^\infty(T_k)} \left\|\zeta_j^\ell\right\|_{L^\infty(T_j)} \frac{c_1 \operatorname{diam}(T_j) \operatorname{diam}(T_k)}{(c_2 \operatorname{dist}(T_j, T_k))^s} (p+1) \left(1 + \frac{2\eta}{c_2}\right) \left(1 + \frac{c_2}{\eta}\right)^{-(p+1)}. \end{aligned} \tag{2.28}$$

Beweis. Der Beweis läuft analog zum Beweis von Satz 2.22. □

3 Lösungsverfahren und numerische Beispiele

3.1 Problemstellung und Lösungsansatz

In diesem Abschnitt betrachten wir die homogene Laplace-Gleichung in der Ebene mit Dirichlet-Randbedingungen

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega \subset \mathbb{R}^2, \\ u &= g \quad \text{auf } \Gamma := \partial\Omega, \end{aligned} \tag{3.1}$$

wobei $\Delta u := \partial_{x_1}^2 u + \partial_{x_1}^2 u$ den Laplace-Operator bezeichnet und $\Omega \subset \mathbb{R}^2$ eine offene Teilmenge mit polygonalem Rand $\Gamma := \partial\Omega$ ist. Wir können weiterhin annehmen, dass $\operatorname{diam}(\Omega) < 1$, da dies durch Skalierung einfach zu erreichen ist.

3.1.1 Randelementmethode

Die Fundamentallösung des Laplace-Operators [9, Lemma 1] ist für $\boldsymbol{z} \in \mathbb{R}^2 \backslash \{0\}$ gegeben durch

$$G(\boldsymbol{z}) := -\frac{1}{2\pi} \log |\boldsymbol{z}|.$$

Mit der Repräsentationsformel [9, Proposition 2] gilt, dass sich die Lösung von (3.1) darstellen lässt als

$$u(\boldsymbol{x}) = \int_\Gamma G(\boldsymbol{x} - \boldsymbol{y}) \partial_{n(\boldsymbol{y})} u(\boldsymbol{y}) ds_{\boldsymbol{y}} - \int_\Gamma \partial_{n(\boldsymbol{y})} G(\boldsymbol{x} - \boldsymbol{y}) u(\boldsymbol{y}) ds_{\boldsymbol{y}}, \tag{3.2}$$

wobei die Integrale über Γ als Oberflächenintegrale zu verstehen sind. Dabei bezeichnet $n(\boldsymbol{y})$ den äußeren Normalenvektor bei $\boldsymbol{y} \in \Gamma$ und $\partial_{n(\boldsymbol{y})}$ die zugehörige Normalenableitung. Weiterhin sei das Einfachschichtpotential $\widetilde{V} \in L\left(H^{-1/2}(\Gamma), H^1(\Omega)\right)$ gegeben durch

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

und das Doppelschichtpotential $\widetilde{K} \in L\left(H^{1/2}(\Gamma), H^1(\Omega)\right)$ durch

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

Für Details hierzu möchten wir auf [9] verweisen.
Mit $\phi(\boldsymbol{y}) = \partial_{n(\boldsymbol{y})} u(\boldsymbol{y})$ und $g(\boldsymbol{y}) = u(\boldsymbol{y})$ lässt sich (3.2) darstellen als

$$u = \widetilde{V}\phi - \widetilde{K}g. \tag{3.3}$$

Sei γ_0 der Spuroperator, der für Funktionen aus $C^\infty(\overline{\Omega})$ durch

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

definiert ist und durch stetige Fortsetzung zu einem Operator $\gamma_0 \in L(H^1(\Omega); H^{1/2}(\Gamma))$ wird. Die schwache Formulierung der Laplace-Gleichung (3.1) lautet dann

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

Die unbekannte schwache Lösung ist dann durch (3.3) gegeben. Um in diese einsetzen zu können, benötigen wir noch die unbekannten Neumann-Randdaten $\phi \in H^{-1/2}(\Gamma)$. Dazu wenden wir den Spuroperator γ_0 auf (3.3) an. Mit dem Einfachschichtpotential

$$V := \gamma_0 \widetilde{V} \in L(H^{-1/2}(\Gamma); H^{1/2}(\Gamma)) \tag{3.4}$$

und dem Doppelschichtpotential[1]

$$K := \gamma_0 \widetilde{K} + 1/2 \in L(H^{1/2}(\Gamma); H^{1/2}(\Gamma)) \tag{3.5}$$

(siehe [9, Kapitel 4]) erhalten wir die Symm'sche Integralgleichung

$$V\phi = (K + 1/2)g. \tag{3.6}$$

Die Variationsformulierung der Symm'schen Integralgleichung lautet

$$\langle V\phi, \psi \rangle_\Gamma = \langle (K + 1/2)g, \psi \rangle_\Gamma \quad \text{für alle } \psi \in H^{-1/2}(\Gamma), \tag{3.7}$$

wobei $\langle \phi, \psi \rangle_\Gamma$ die duale Paarung zwischen $H^{1/2}(\Gamma) \times H^{-1/2}(\Gamma)$ bezeichnet, die für $\phi, \psi \in L^2(\Gamma)$ zu $\langle \phi, \psi \rangle_\Gamma = \int_\Gamma \phi\psi \, ds_{\boldsymbol{x}}$ vereinfacht werden kann. Laut [9, Kapitel 4] ist der Operator V unter der Annahme $\operatorname{diam}(\Omega) < 1$ ein symmetrischer und elliptischer Isomorphismus. Zusätzlich definiert die linke Seite von (3.7)

$$\langle\langle \phi, \psi \rangle\rangle_V := \langle V\phi, \psi \rangle_\Gamma \quad \text{für alle } \phi, \psi \in H^{-1/2}(\Gamma) \tag{3.8}$$

ein äquivalentes Skalarprodukt auf $H^{-1/2}(\Gamma)$. Dazu bezeichne $|||\cdot|||$ die induzierte Norm. Laut [11, Kapitel 4.1] hat die Variationsformulierung der Symm'schen Integralgleichung eine eindeutige Lösung $\phi \in H^{-1/2}(\Gamma)$.

3.1.2 Galerkin-Verfahren

Da das Problem (3.7) im Allgemeinen nicht exakt lösbar ist, werden wir es approximativ mit Hilfe des Galerkin-Verfahrens lösen. Dabei wird $H^{-1/2}(\Gamma)$ beziehungsweise $H^{1/2}(\Gamma)$ durch einen endlichdimensionalen Unterraum ersetzt.
Wir betrachten eine Partition $\mathcal{T}_h = \{T_1, T_2, \ldots, T_N\}$ des Randes Γ in N affine Randstücke. Im folgenden bezeichnen wir $\mathcal{T}_h$ auch als *Triangulierung* von Γ. Sei $\mathcal{P}^p(\mathcal{T}_h)$ der Raum aller $\mathcal{T}_h$-stückweisen Polynome vom Grad $p \in \mathbb{N}_0$. Die Funktionen aus $\mathcal{P}^p(\mathcal{T}_h)$ sind im Allgemeinen nicht stetig, sondern haben Sprünge an den Knotenpunkten von $\mathcal{T}_h$. Insbesondere ist dabei der Raum $\mathcal{P}^0(\mathcal{T}_h)$ der Raum der $\mathcal{T}_h$-stückweise konstanten Funktionen und $\mathcal{P}^1(\mathcal{T}_h)$ der Raum der $\mathcal{T}_h$-stückweise affinen Funktionen. Bezeichne $\chi_j^0 \in \mathcal{P}^0(\mathcal{T}_h)$ die charakteristische Funktion von $T_j \in \mathcal{T}_h$ definiert durch

$$\chi_j^0(\boldsymbol{x}) = \begin{cases} 1 & \text{für } \boldsymbol{x} \in T_j \\ 0 & \text{sonst} \end{cases}$$

für alle $j = 1, \ldots, N$. Dann ist die Menge $\{\chi_1^0, \chi_2^0, \ldots, \chi_N^0\}$ eine Basis von $\mathcal{P}^0(\mathcal{T}_h)$. Um eine Basis von $\mathcal{P}^1(\mathcal{T}_h)$ angeben zu können benötigen wir zusätzlich noch $\mathcal{T}_h$-stückweise lineare Funktionen. Dazu sei $\chi_j^1 \in \mathcal{P}^1(\mathcal{T}_h)$ über die Parametrisierung γ_j auf T_j definiert durch

$$\left(\chi_j^1 \circ \gamma_j\right)(s) = s \quad \text{für } s \in [-1, 1].$$

[1] In der Literatur werden sowohl $\widetilde{V}$ als auch V als Einfachschichtpotential und sowohl $\widetilde{K}$ als auch K als Doppelschichtpotential bezeichnet.

Das heißt für $T_j = [\boldsymbol{a}_j, \boldsymbol{b}_j]$ ist χ_j^1 eine lineare Funktion, die am Punkt $\boldsymbol{a}_j$ den Wert -1 und am Punkt $\boldsymbol{b}_j$ den Wert 1 annimmt. Auf allen anderen affinen Randstücken $T_k \in \mathcal{T}_h$ mit $k = 1, \dots, N$ und $k \neq j$ sei $\chi_j^1 \equiv 0$. Dann ist die Menge $\{\chi_1^0, \chi_2^0, \dots, \chi_N^0, \chi_1^1, \chi_2^1, \dots, \chi_N^1\}$ eine Basis von $\mathcal{P}^1(\mathcal{T}_h)$.

Um auch global stetige Funktionen betrachten zu können, sei $\mathcal{S}^p(\mathcal{T}_h) := \mathcal{P}^p(\mathcal{T}_h) \cap C(\Gamma)$ der Raum aller global stetigen und $\mathcal{T}_h$-stückweise polynomialen Funktionen vom Grad p. Für die Menge der Knoten $\mathcal{K}_h = \{\boldsymbol{z}_1, \dots, \boldsymbol{z}_{\widetilde{N}}\}$, wobei $\widetilde{N}$ die Anzahl der Knoten bezeichnet[2], sei jedem $\boldsymbol{z}_j \in \mathcal{K}_h$ eine nodale Hutfunktion $\zeta_j^1 \in \mathcal{S}^1(\mathcal{T}_h)$ zugeordnet. Für diese stückweise affine Funktion gelte also $\zeta_j^1(\boldsymbol{z}_k) = \delta_{jk}$ für alle $k = 1, \dots, N$. Der Knoten $\boldsymbol{z}_j$ ist im Allgemeinen Randknoten von zwei Elementen: T_{j_1} und T_{j_2}. Diese Elemente und ihre Parametrisierungen γ_{j_1} und γ_{j_2} seien so gewählt, dass

$$\begin{aligned} (\zeta_j^1 \circ \gamma_{j_1})(t) &= \frac{1+t}{2} \\ (\zeta_j^1 \circ \gamma_{j_2})(t) &= \frac{1-t}{2} \end{aligned} \tag{3.9}$$

für $t \in [-1, 1]$ gilt. Die Menge der $\zeta_j^1, j = 1, \dots, \widetilde{N}$ bildet bereits eine Basis von $\mathcal{S}^1(\mathcal{T}_h)$.

Um nun eine Basis von $\mathcal{S}^2(\mathcal{T}_h)$ angeben zu können, benötigen wir noch $\mathcal{T}_h$-stückweise quadratische Terme. Wir definieren $\zeta_j^2 \in \mathcal{S}^2(\mathcal{T}_h)$ über die Parametrisierung γ_j auf T_j:

$$\left(\zeta_j^2 \circ \gamma_j\right)(s) = \frac{1}{4}(1 - s^2) \quad \text{für } s \in [-1, 1]. \tag{3.10}$$

Auf allen anderen affinen Randstücken $T_k \in \mathcal{T}_h$ mit $k = 1, \dots, N$ und $k \neq j$ sei $\zeta_j^2 \equiv 0$. Die Menge $\{\zeta_1^1, \zeta_2^1, \dots, \zeta_{\widetilde{N}}^1, \zeta_1^2, \zeta_2^2, \dots, \zeta_N^2\}$ stellt dann eine Basis von $\mathcal{S}^2(\mathcal{T}_h)$ dar.

Um (3.7) zu diskretisieren approximieren wir den Raum $H^{1/2}(\Gamma)$ mit dem endlichdimensionalen Teilraum $\mathcal{S}^2(\mathcal{T}_h) \subset H^{1/2}(\Gamma)$ und den Raum $H^{-1/2}(\Gamma)$ mit dem endlichdimensionalen Teilraum $\mathcal{P}^1(\mathcal{T}_h) \subset H^{-1/2}(\Gamma)$.

Weiters diskretisieren wir $g \in H^{1/2}$ mit einer Funktion aus $\mathcal{S}^2(\mathcal{T}_h)$. Formal ist $g \in H^{1/2}(\Gamma)$, allerdings nehmen wir zusätzlich an, dass $g \in H^1(\Gamma) \subset H^{1/2}(\Gamma)$, wodurch wir erreichen, dass g zusätzlich stetig ist. Eine Möglichkeit ist, g durch seine L^2-Projektion auf $\mathcal{S}^2(\mathcal{T}_h)$ zu diskretisieren

$$G_h := \sum_{j=1}^{\widetilde{N}} g_{h;j}^1 \zeta_j^1 + \sum_{j=1}^{N} g_{h;j}^2 \zeta_j^2 \in \mathcal{S}^2(\mathcal{T}_h) \subset H^1(\Gamma) \tag{3.11}$$

mit

$$\int_\Gamma G_h \zeta \, d\Gamma = \int_\Gamma g \zeta \, d\Gamma \quad \text{für alle } \zeta \in \mathcal{S}^2(\mathcal{T}_h), \tag{3.12}$$

wobei $g_{h;j}^k \in \mathbb{R}$ die entsprechenden Koeffizienten des Koeffizientenvektors $\boldsymbol{g}_h = (\boldsymbol{g}_h^1, \boldsymbol{g}_h^2)^T \in \mathbb{R}^{\widetilde{N}+N}$ sind. Da der diskrete Raum $\mathcal{P}^1(\mathcal{T}_h)$ ein Teilraum von $H^{1/2}(\Gamma)$ ist, definiert $\langle\!\langle \cdot, \cdot \rangle\!\rangle_V$ auch ein Skalarprodukt auf $\mathcal{P}^1(\mathcal{T}_h)$. Daher gibt es eine eindeutige Lösung $\Phi_h \in \mathcal{P}^1(\mathcal{T}_h)$ von

$$\langle V\Phi_h, \Psi_h \rangle_\Gamma = \langle (K + 1/2)\, G_h, \Psi_h \rangle_\Gamma \quad \text{für alle } \Psi_h \in \mathcal{P}^1(\mathcal{T}_h).$$

[2] Für geschlossene Ränder Γ gilt, dass die Anzahl der Knoten mit der Anzahl an Elementen übereinstimmt, das heißt es gilt $\widetilde{N} = N$.

Dies ist äquivalent zu

$$\langle V\Phi_h, \chi_i^m\rangle_\Gamma = \langle (K+1/2)\, G_h, \chi_i^m\rangle_\Gamma \quad \text{für alle } i = 1,\ldots,N \text{ und } m = 0,1. \tag{3.13}$$

Sei $\boldsymbol{x}_h = (\boldsymbol{x}_h^0, \boldsymbol{x}_h^1)^T \in \mathbb{R}^{2N}$ der Koeffizientenvektor für den Ansatz

$$\Phi_h = \sum_{j=1}^{N} x_{h;j}^0 \chi_j^0 + \sum_{j=1}^{N} x_{h;j}^1 \chi_j^1.$$

Nun können wir (3.13) umschreiben:

$$\sum_{k=0}^{1}\sum_{j=1}^{N} x_{h;j}^k \langle V\chi_j^k, \chi_i^m\rangle_\Gamma = \sum_{k=1}^{2}\sum_{j=1}^{N_k} g_{h;j}^k \langle (K+1/2)\, \zeta_j^1, \chi_i^m\rangle_\Gamma \tag{3.14}$$

für alle $i = 1,\ldots,N$ und $m = 0,1$. Dabei ist $N_1 = \widetilde{N}$ und $N_2 = N$. Für das Einfachschichtpotential V sei die Einfachschichtpotential-Operatormatrix $\mathbf{V}$ für $\mathcal{P}^1(\mathcal{T}_h)$ Ansatz- und Testfunktionen definiert durch

$$\mathbf{V} := \begin{pmatrix} \mathbf{V}^{(00)} & \mathbf{V}^{(01)} \\ \mathbf{V}^{(10)} & \mathbf{V}^{(11)} \end{pmatrix} \in \mathbb{R}^{2N\times 2N}, \tag{3.15}$$

wobei die einzelnen Submatrizen $\mathbf{V}^{(\ell m)} \in \mathbb{R}^{N\times N}$ für $\ell, m \in \{0,1\}$ durch

$$\begin{aligned} \mathbf{V}_{ij}^{(mk)} &:= \langle V\chi_j^k, \chi_i^m\rangle_\Gamma \\ &= -\frac{1}{2\pi}\int_{T_i}\int_{T_j} \log\left(|\boldsymbol{x}-\boldsymbol{y}|\right) \chi_j^k(\boldsymbol{y})\chi_i^m(\boldsymbol{x}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \end{aligned} \tag{3.16}$$

gegeben sind. Analog dazu kann man auch für den Doppelschichtoperator K die Doppelschicht-Operatormatrix $\mathbf{K}$ für $\mathcal{S}^2(\mathcal{T}_h)$ Ansatz- und $\mathcal{P}^1(\mathcal{T}_h)$ Testfunktionen definieren:

$$\mathbf{K} := \begin{pmatrix} \mathbf{K}^{(01)} & \mathbf{K}^{(02)} \\ \mathbf{K}^{(11)} & \mathbf{K}^{(12)} \end{pmatrix} \in \mathbb{R}^{2N\times(\widetilde{N}+N)}. \tag{3.17}$$

Die Einträge der Submatrizen $\mathbf{K}^{(m1)} \in \mathbb{R}^{N\times\widetilde{N}}$ und $\mathbf{K}^{(m2)} \in \mathbb{R}^{N\times N}$ für $m \in \{0,1\}$ sind dabei durch

$$\begin{aligned} \mathbf{K}_{ij}^{(mk)} &:= \langle K\zeta_j^k, \chi_i^m\rangle_\Gamma \\ &= -\frac{1}{2\pi}\int_{T_i}\int_{\operatorname{supp}(\zeta_j^m)} \frac{(\boldsymbol{y}-\boldsymbol{x})\cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2} \zeta_j^k(\boldsymbol{y})\chi_i^m(\boldsymbol{x}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \end{aligned} \tag{3.18}$$

gegeben. Für $k = 2$ stimmt der Träger $\operatorname{supp}(\zeta_j^k)$ mit T_j überein. Im Fall $k = 1$ erstreckt sich der Träger von ζ_j^k über bis zu zwei Elemente. Das sind genau die beiden, bei welchen $\boldsymbol{z}_j$ ein Eckpunkt ist.
Zusätzlich möchten wir noch die Masse-Matrix $\mathbf{M} \in \mathbb{R}^{2N\times(\widetilde{N}+N)}$ definieren:

$$\mathbf{M} := \begin{pmatrix} \mathbf{M}^{(01)} & \mathbf{M}^{(02)} \\ \mathbf{M}^{(11)} & \mathbf{M}^{(12)} \end{pmatrix} \in \mathbb{R}^{2N\times(\widetilde{N}+N)}. \tag{3.19}$$

Die Einträge der Submatrizen $\mathbf{M}^{(m1)} \in \mathbb{R}^{N\times\widetilde{N}}$ für $m \in \{0,1\}$ und $\mathbf{M}^{(m2)} \in \mathbb{R}^{N\times N}$ für $m \in \{0,1\}$ sind dabei durch

$$\mathbf{M}_{ij}^{(mk)} := \langle \zeta_j^k, \chi_i^m \rangle_\Gamma \tag{3.20}$$

gegeben.
Nun können wir die Galerkin Formulierung (3.14) äquivalent als lineares Gleichungssystem darstellen:

$$\mathbf{V}\boldsymbol{x}_h = \mathbf{K}\boldsymbol{g}_h + \frac{1}{2}\mathbf{M}\boldsymbol{g}_h. \tag{3.21}$$

Insbesondere ist $\mathbf{V}$ eine positiv definite symmetrische Matrix, da sie von einem Skalarprodukt, nämlich $\mathbf{V}_{ij}^{(mk)} = \langle\!\langle \chi_j^k, \chi_i^m \rangle\!\rangle_V$, abgeleitet ist.

Bemerkung 3.1. *Um die Resultate aus Abschnitt 2 auf die Matrizen $\boldsymbol{V}$ und $\boldsymbol{K}$ anwenden zu können, ist es notwendig, sich zu überlegen, dass* $\log(|\boldsymbol{x}-\boldsymbol{y}|)$ *und* $(\boldsymbol{y}-\boldsymbol{x})/|\boldsymbol{x}-\boldsymbol{y}|^2$ *für* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$ *(komponentenweise) asymptotisch glatte Kernfunktionen sind.*
Für die eindimensionale Kernfunktion $(x,y) \mapsto \log(|x-y|)$ *mit* $x, y \in \mathbb{R}$ *sieht man mit Hilfe von [15, Gleichung (4.12)]*

$$\partial_x^\ell \log(|x-y|) = (-1)^{\ell-1}\frac{(\ell-1)!}{(x-y)^\ell} \qquad \textit{für alle } \ell \in \mathbb{N}$$

unmittelbar, dass es sich um eine asymptotisch glatte Kernfunktion handelt. Dabei sei eine eindimensionale asymptotisch glatte Kernfunktion analog zu Definition 2.14 erklärt. [15, Satz E.2.1] verallgemeinert dieses Resultat nun auch auf die entsprechende zweidimensionale Funktion $(\boldsymbol{x}, \boldsymbol{y}) \mapsto log(|\boldsymbol{x}-\boldsymbol{y}|)$ *für* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$.
Die Funktion $(\boldsymbol{x}, \boldsymbol{y}) \mapsto (\boldsymbol{y}-\boldsymbol{x}) \cdot n(\boldsymbol{y})/|\boldsymbol{x}-\boldsymbol{y}|^2$ *ist genau die Normalenableitung von* $\log(|\boldsymbol{x}-\boldsymbol{y}|)$ *entlang von* $n(\boldsymbol{y})$. *Mit Definition 2.14 sehen wir, dass die Ableitung einer asymptotisch glatten Kernfunktion wieder asymptotisch glatt ist, das heißt* $\partial_z \log(|\boldsymbol{x}-\boldsymbol{y}|)$ *für* $z \in \{y_1, y_2\}$ *ist asymptotisch glatt. Es gilt*

$$\frac{(\boldsymbol{y}-\boldsymbol{x}) \cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2} = \partial_{y_1} \log(|\boldsymbol{x}-\boldsymbol{y}|)n_1(\boldsymbol{y}) + \partial_{y_2} \log(|\boldsymbol{x}-\boldsymbol{y}|)n_2(\boldsymbol{y}) \tag{3.22}$$

für $n(\boldsymbol{y}) = (n_1(\boldsymbol{y}), n_2(\boldsymbol{y}))^T \in \mathbb{R}^2$. *Sei nun* $\boldsymbol{y} \in T_j$ *für ein festes* T_j. *Dann ist* $n(\boldsymbol{y})$ *konstant. Für einen Eintrag der Matrix* $\boldsymbol{K}$ *betrachten wir den Integranden (siehe* (3.18)*)*

$$\frac{(\boldsymbol{y}-\boldsymbol{x}) \cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2}\zeta_j^k(\boldsymbol{y})\chi_i^m(\boldsymbol{x}).$$

Wenn wir nun definieren $\widetilde{\zeta_j^k}(\boldsymbol{y}) := n_1(\boldsymbol{y})\zeta_j^k(\boldsymbol{y})$ *und diese Funktion statt* $\zeta_j^k(\boldsymbol{y})$ *in* (3.18) *einsetzen, so haben wir mit*

$$\partial_{y_1} \log(|\boldsymbol{x}-\boldsymbol{y}|) = (y_1 - x_1)/(|\boldsymbol{x}-\boldsymbol{y}|^2)$$

eine asymptotisch glatte Kernfunktion und können die Resultate aus Abschnitt 2 anwenden. Analog verfährt man für die zweite Komponente.

Wie bereits in Bemerkung 2.13 erwähnt, betrachten wir nicht genau die Matrix $\mathbf{V}$, sondern eine approximierende Matrix $\mathbf{V}_p$. Bei dieser wird bei bestimmten Einträgen das Integral über das kürzere Randstück durch Gauß-Quadratur vom Grad p ersetzt. In anderen Fällen werden beide Integrale durch Quadratur ersetzt. Für Details möchten wir hier auf Abschnitt 2 verweisen. Da bei der analytischen Berechnung der Einträge der Matrix $\mathbf{V}$ Auslöschungseffekte auftreten, ist es praktisch gesehen sinnvoller, für eine stabile Berechnung die approximierende Matrix $\mathbf{V}_p$ zu verwenden. In Satz 2.22 beziehungsweise Satz 2.24 haben wir gezeigt, dass die Matrix $\mathbf{V}_p$ in der Frobeniusnorm exponentiell schnell gegen die Matriv $\mathbf{V}$ konvergiert. Für diesen Abschnitt möchten wir im folgenden nicht mehr unterscheiden, auf welche Weise $\mathbf{V}$ berechnet wurde, sondern schreiben auch für die approximierende Matrix einfach $\mathbf{V}$.

Auch für die Matrix $\mathbf{K}$ möchten wir eine approximierende Matrix $\mathbf{K}_p$ verwenden. Ähnlich wie in Definition 2.21 möchten wir bei η-max-Zulässigkeit beide Integrale durch Gauß-Quadratur vom Grad p ersetzten und bei η-min-Zulässigkeit nur das Integral über das kürzere Randstück durch Quadratur ersetzen. Eine zweite Variante wäre, die Matrix $\mathbf{K}$ durch die approximierende Matrix $\mathbf{K}_p$ gemäß Definition 2.23 zu definieren.
In beiden Fällen müssen wir uns überlegen, dass bei $\mathbf{K}$ Ansatzfunktionen auftreten, deren Träger sich im Allgemeinen über mehr als ein Element, maximal aber über zwei Elemente, erstrecken. Dies betrifft allerdings ausschließlich die Basisfunktionen ζ_j^1 für $j = 1, \ldots, \widetilde{N}$ des Ansatzraums $\mathcal{S}^1(\mathcal{T}_h)$. Für die stückweise quadratischen Basisfunktionen ζ_j^2 für $j = 1, \ldots, N$ gilt $\operatorname{supp}(\zeta_j^2) \subset T_j$. Sei für einen festen Eintrag $\mathbf{K}_{ij}^{(m1)}$ der Matrix $\mathbf{K}$ die zugehörige Basisfunktion $\zeta_j^1 \in \mathcal{S}^1(\mathcal{T}_h)$, die zu einem bestimmten Knoten $\boldsymbol{z}_j$ gehört. Dieser Knoten ist im Allgemeinen Randknoten von zwei Elementen. Diese bezeichnen wir mit T_{j_1} und T_{j_2}. Dann können wir (3.18) für $m = 1$ auch schreiben als

$$\begin{aligned} \mathbf{K}_{ij}^{(m1)} &= -\frac{1}{2\pi} \int_{T_i} \int_{T_{j_1}} \frac{(\boldsymbol{y} - \boldsymbol{x}) \cdot n(\boldsymbol{y})}{|\boldsymbol{x} - \boldsymbol{y}|^2} \zeta_j^1(\boldsymbol{y}) \chi_i^m(\boldsymbol{x}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &\quad - \frac{1}{2\pi} \int_{T_i} \int_{T_{j_2}} \frac{(\boldsymbol{y} - \boldsymbol{x}) \cdot n(\boldsymbol{y})}{|\boldsymbol{x} - \boldsymbol{y}|^2} \zeta_j^1(\boldsymbol{y}) \chi_i^m(\boldsymbol{x}) ds_{\boldsymbol{y}} ds_{\boldsymbol{x}}. \end{aligned} \tag{3.23}$$

Das heißt, die Elemente $\mathbf{K}_{ij}^{(m1)}$ der Matrix $\mathbf{K}$ lassen sich auch als Summe zweier Doppelintegrale darstellen, wobei der Integrationsbereich der einzelnen Integrale sich in diesem Fall nur über je ein Element erstreckt. Nun können wir so wie in Definition 2.21 beziehungsweise 2.23 vorgehen und für die einzelnen Doppelintegrale getrennt die Zulässigkeitsbedingungen überprüfen und so die Approximation für die Doppelintegrale berechnen. Mit dieser Methode erhalten wir auch eine approximierende Matrix $\mathbf{K}_p$ für $\mathbf{K}$.
Ebenso wie für $\mathbf{V}$ erhalten wir auch für $\mathbf{K}$ mit Satz 2.22 beziehungsweise Satz 2.24, dass die approximierende Matrix exponentiell schnell gegen die exakte Matrix $\mathbf{K}$ konvergiert. Auch für $\mathbf{K}$ möchten wir für diesen Abschnitt notationell nicht mehr unterscheiden auf welche Weise $\mathbf{K}$ berechnet wurde und schreiben sowohl für die exakte Matrix, die approximierende Matrix nach max-min-Kriterium als auch für die approximierende Matrix nach min-Kriterium einfach $\mathbf{K}$.

3.1.3 Indirekte Randintegralmethode

Eine andere Methode zur Lösung von (3.1) ist die indirekte Randintegralmethode. Mit dem Einfachschichtpotential $\widetilde{V}$ gilt nun laut [9, Theorem 12]

$$-\Delta \widetilde{V}\phi = 0 \quad \text{für alle } \phi \in H^{-1/2}(\Gamma). \tag{3.24}$$

Nun können wir den sogenannten *indirekten Ansatz* $u = \widetilde{V}\phi$ machen. Mit (3.24) ist die Laplace-Gleichung erfüllt und es gelte für gegebenes $g \in H^{1/2}(\Gamma)$

$$V\phi = g, \tag{3.25}$$

wobei $V = \gamma_0 \widetilde{V}$ wie in (3.4). Wenn man nun eine Funktion $\phi \in H^{-1/2}(\Gamma)$ finden kann, die (3.25) löst, dann hat man mit $\widetilde{V}\phi$ eine Lösung des Problems (3.1) gefunden. Mit $\langle\!\langle \cdot, \cdot \rangle\!\rangle_V$ aus (3.8) erhalten wir die Variationsformulierung von (3.25):

$$\langle\!\langle \phi, \psi \rangle\!\rangle_V = \langle g, \psi \rangle_\Gamma \quad \text{für alle } \psi \in H^{-1/2}(\Gamma). \tag{3.26}$$

Für dieses Problem existiert zwar eine eindeutige Lösung, da es aber im Allgemeinen nicht möglich ist, diese exakt zu berechnen, werden wir auch hier (3.26) approximativ unter Anwendung des Galerkin-Verfahrens lösen. Wie in Kapitel 3.1.2 approximieren wir auch hier den Raum $H^{-1/2}(\Gamma)$ durch den endlichdimensionalen Unterraum $\mathcal{P}^1(\mathcal{T}_h)$ und den Raum $H^{1/2}(\Gamma)$ durch den endlichdimensionalen Unterraum $\mathcal{S}^2(\mathcal{T}_h)$. Dabei diskretisieren wir zuerst $g \in H^{1/2}(\Gamma)$ indem wir g durch die L^2-Projektion G_h auf $\mathcal{S}^2(\mathcal{T}_h)$ approximieren. Sei also G_h so wie in (3.11) gemeinsam mit der Bedingung (3.12). Dann suchen wir eine Lösung $\Phi_h \in \mathcal{P}^1(\mathcal{T}_h)$ mit

$$\langle\!\langle \Phi_h, \psi_h \rangle\!\rangle_V = \langle G_h, \psi_h \rangle_\Gamma \quad \text{für alle } \psi_h \in \mathcal{P}^1(\mathcal{T}_h).$$

Mit der Basis $\{\chi_1^0, \chi_2^0, \ldots, \chi_N^0, \chi_1^1, \chi_2^1, \ldots, \chi_N^1\}$ von $\mathcal{P}^1(\mathcal{T}_h)$ ist das äquivalent zu

$$\langle\!\langle \Phi_h, \chi_i^m \rangle\!\rangle_V = \langle G_h, \chi_i^m \rangle_\Gamma \quad \text{für alle } i = 1, \ldots, N \text{ und } m = 0, 1. \tag{3.27}$$

Da $\langle\!\langle \cdot, \cdot \rangle\!\rangle_V$ auch ein Skalarprodukt auf $\mathcal{P}^1(\mathcal{T}_h) \subset H^{-1/2}(\Gamma)$ ist, existiert für (3.27) eine eindeutige Lösung $\Phi_h \in \mathcal{P}^1(\mathcal{T}_h)$. Die Galerkin-Lösung ist sogar die Bestapproximation bezüglich des Unterraums $\mathcal{P}^1(\mathcal{T}_h)$. Das heißt, es gilt

$$|\!|\!| \phi - \Phi_h |\!|\!| \leq |\!|\!| \phi - \psi_h |\!|\!| \quad \text{für alle } \psi_h \in \mathcal{P}^1(\mathcal{T}_h).$$

Sei nun $\boldsymbol{x}_h = (\boldsymbol{x}_h^0, \boldsymbol{x}_h^1)^T \in \mathbb{R}^{2N}$ der Koordinatenvektor der diskreten Lösung Φ_h bezüglich der Basis $\{\chi_1^0, \chi_2^0, \ldots, \chi_N^0, \chi_1^1, \chi_2^1, \ldots, \chi_N^1\}$. Das heißt, es gilt

$$\Phi_h = \sum_{j=1}^{N} \boldsymbol{x}_{h;j}^0 \chi_j^0 + \sum_{j=1}^{N} \boldsymbol{x}_{h;j}^1 \chi_j^1.$$

Nun können wir (3.27) folgendermaßen umschreiben:

$$\sum_{k=0}^{1} \sum_{j=1}^{N} \boldsymbol{x}_{h;j}^k \langle V\chi_j^k, \chi_i^m \rangle_\Gamma = \sum_{k=1}^{2} \sum_{j=1}^{N_k} g_{h;j}^k \langle \zeta_j^k, \chi_i^m \rangle_\Gamma \quad \text{für alle } i = 1, \ldots, N \text{ und } m = 0, 1. \tag{3.28}$$

Dabei ist $N_1 = \widetilde{N}$ und $N_2 = N$ und $g^k_{h;j} \in \mathbb{R}$ sind die entsprechenden Koeffizienten des Koeffizientenvektors $\boldsymbol{g}_h = (\boldsymbol{g}^1_h, \boldsymbol{g}^2_h)^T \in \mathbb{R}^{\widetilde{N}+N}$ aus der Darstellung (3.11). Mit der Matrix $\mathbf{V}$ aus (3.15) beziehungsweise (3.16) und $\mathbf{M}$ aus (3.19) beziehungsweise (3.20) können wir nun (3.28) folgendermaßen umschreiben:

$$\mathbf{V}\boldsymbol{x}_h = \mathbf{M}\boldsymbol{g}_h.$$

Auch hier verwenden wir eigentlich nicht die Matrix $\mathbf{V}$, sondern die approximierende Matrix $\mathbf{V}_p$ und unterscheiden auch im Weiteren nicht auf welche Weise $\mathbf{V}$ berechnet wurde und schreiben auch für die approximierende Matrix $\mathbf{V}$.

Bemerkung 3.2. *Sowohl für die Randelementmethode als auch für die indirekte Randintegralmethode haben wir nun als Diskretisierungsräume für $H^{1/2}(\Gamma)$ den Unterraum $\mathcal{S}^2(\mathcal{T}_h)$ und für $H^{-1/2}(\Gamma)$ den Unterraum $\mathcal{P}^1(\mathcal{T}_h)$ gewählt. Stattdessen hätten wir auch $\mathcal{S}^1(\mathcal{T}_h) \subset H^{1/2}(\Gamma)$ beziehungsweise $\mathcal{P}^0(\mathcal{T}_h) \subset H^{-1/2}(\Gamma)$ wählen können. Das Galerkin-Verfahren lässt sich hier sowohl für die Randelementmethode als auch für die indirekte Randintegralmethode mit Vereinfachungen an einigen Stellen komplett analog durchführen. Der Vorteil dabei ist, dass sich die Dimension der auftretenden Vektoren und Matrizen verringert, da auch die Dimension der Diskretisierungsräume geringer ist. Für den Koeffizientenvektor $\boldsymbol{g}_h$ der diskretisierten Dirichletdaten G_h beziehungsweise den Koeffizientenvektor $\boldsymbol{x}_h$ der diskreten Lösung Φ_h gilt $\boldsymbol{g}_h, \boldsymbol{x}_h \in \mathbb{R}^N$. Auch die Matrizen $\boldsymbol{V}, \boldsymbol{K}$ und $\boldsymbol{M}$ reduzieren sich auf die erste Submatrix $\boldsymbol{V}^{(00)} \in \mathbb{R}^{N\times N}$ und $\boldsymbol{K}^{(01)}, \boldsymbol{M}^{(01)} \in \mathbb{R}^{N\times\widetilde{N}}$ (vgl.* (3.15),(3.17) *und* (3.19)*). Als Folge davon müssen wir bei den Gleichungen $\boldsymbol{V}\boldsymbol{x}_h = \boldsymbol{K}\boldsymbol{g}_h + 1/2\boldsymbol{M}\boldsymbol{g}_h$ beziehungsweise $\boldsymbol{V}\boldsymbol{x}_h = \boldsymbol{g}_h$ nur N und nicht $2N$ lineare Gleichungen lösen. Für eine große Elementanzahl N ist das natürlich weniger aufwendig.*
Für die $\mathcal{S}^1(\mathcal{T}_h)$-$\mathcal{P}^0(\mathcal{T}_h)$-Variante ist das Galerkin-Verfahren und die Berechnung der Fehlerschätzer in [12] ausführlich dokumentiert.
Der Fokus dieser Arbeit liegt allerdings auf der $\mathcal{S}^2(\mathcal{T}_h)$-$\mathcal{P}^1(\mathcal{T}_h)$-Variante. Obwohl diese Variante aufwendiger ist, ist die Galerkin-Approximation der exakten Lösung wesentlich besser.

3.1.4 Fehlerschätzer

Mit Hilfe des Galerkin-Verfahrens wird nur eine approximative Lösung für das Problem ermittelt. Damit wir nun eine Aussage über die Genauigkeit der approximierten Lösung machen können, interessieren wir uns für den Fehler $|||\phi - \Phi_h|||$.

Für die Randelementmethode sei ϕ die exakte Lösung von (3.7) und Φ_h die zugehörige Galerkin-Lösung. Zusätzlich sei ϕ_h die exakte Lösung der Variationsformulierung

$$\langle V\phi_h, \psi\rangle_\Gamma = \langle (K + 1/2)G_h, \psi\rangle_\Gamma \quad \text{für alle } \psi \in H^{-1/2}(\Gamma) \tag{3.29}$$

mit gestörter rechter Seite, bei welcher wir die Approximation $G_h \approx g$ verwenden. Mit dem Galerkin-Verfahren approximieren wir eigentlich nicht die Lösung von (3.7), sondern die Lösung ϕ_h von (3.29). Betrachten wir also den Fehler $|||\phi_h - \Phi_h|||$. Da es im Allgemeinen nicht möglich ist die Lösung ϕ_h zu ermitteln, möchten wir einen Fehlerschätzer finden, der sich im Wesentlichen so verhält wie der Fehler selbst. Da das Einfachschichtpotential $V : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ ein Isomorphismus ist, können wir nun das Residuum der Symm'schen Integralgleichung (3.6) auf dem Netz $\mathcal{T}_h$ betrachten:

$$\|V\Phi_h - (K + 1/2)G_h\|_{H^{1/2}(\Gamma)} \simeq |||\phi_h - \Phi_h|||.$$

Die linke Seite der oberen Gleichung liefert uns also einen effizienten und zuverlässigen Fehlerschätzer für die rechte Seite. Da wir später den Fehlerschätzer verwenden wollen, um das Netz $\mathcal{T}_h$ lokal zu verfeinern, brauchen wir einen Fehlerschätzer mit lokalen Beiträgen für jedes Element $T \in \mathcal{T}_h$. Da uns die $H^{1/2}$-Norm keine lokale Information liefert, können wir laut [12, Kapitel 9.7] auch die folgende Lokalisierung verwenden, für welche gilt

$$\begin{aligned} &\|V\Phi_h - (K+1/2)G_h\|^2_{H^{1/2}(\Gamma)} \\ &\quad \lesssim \sum_{T\in\mathcal{T}_h} \operatorname{diam}(T)\,\|(V\Phi_h - (K+1/2)G_h)'\|^2_{L^2(T)} =: \sum_{T\in\mathcal{T}_h} \rho_h(T)^2 =: \rho_h^2. \end{aligned} \tag{3.30}$$

Dabei bezeichne $(\cdot)'$ die Tangentialableitung. Das Zeichen $\lesssim$ bedeutet, dass die linke Seite kleiner gleich einer positiven Konstante mal der rechten Seite ist. In unserem Fall hängt diese Konstante nur von Γ und einer oberen Schranke für die *K-Netz Konstante*

$$\kappa(\mathcal{T}_h) := \sup\{\operatorname{diam}(T_j)/\operatorname{diam}(T_k) : T_j, T_k \in \mathcal{T}_h \text{ mit } T_j \cap T_k \neq \emptyset\}$$

ab. Als *Residualschätzer* ρ_h bezeichnen wir nun den globalen Fehlerschätzer, welcher als $\rho_h = \left\|h^{1/2}(V\Phi_h - (1/2+K)G_h)'\right\|_{L^2(\Gamma)}$ geschrieben werden kann. Hier bezeichne $h \in \mathcal{P}^0(\mathcal{T}_h)$ die *lokale Netzweitenfunktion*, eine stückweise konstante Funktion, die auf jedem Element die Länge dieses Elements annimmt. Das heißt, es gilt $h|_T = \operatorname{diam}(T)$ für alle $T \in \mathcal{T}_h$.

Auch für die indirekte Randintegralmethode können wir analog zur Randelementmethode vorgehen und erhalten auch hier einen gewichteten Residualschätzer

$$\rho_h := \left\|h^{1/2}(V\Phi_h - G_h)'\right\|_{L^2(\Gamma)}. \tag{3.31}$$

Unter Anwendung von [14, Theorem 5.1] in Kombination mit [14, Theorem 7.2] erhalten wir, dass sowohl ρ_h aus (3.30) als auch ρ_h aus (3.31) zuverlässige Fehlerschätzer sind.

3.1.5 Netzverfeinerung

Um den Approximationsfehler zu verbessern, gibt es die Möglichkeit, das Gitternetz $\mathcal{T}_h$ zu verfeinern. Das heißt, wir wählen ein feineres Gitter $\widehat{\mathcal{T}}_h$, sodass $\mathcal{P}^1(\mathcal{T}_h) \subset \mathcal{P}^1(\widehat{\mathcal{T}}_h)$. Damit wird der Fehler wegen

$$|||\phi - \widehat{\Phi}_h||| \leq |||\phi - \widehat{\Psi}_h||| \quad \text{für alle } \widehat{\Psi}_h \in \mathcal{P}^1(\widehat{\mathcal{T}}_h)$$

kleiner. Dabei sei $\widehat{\Phi}_h$ die Galerkin-Lösung auf dem feineren Gitter.
Für die Netzverfeinerung gibt es mehrere Möglichkeiten. Eine Option ist, das Netz *uniform* zu verfeinern, indem man in jedem Verfeinerungsschritt jedes Element in eine feste Anzahl von feineren Elementen unterteilt. Für diese Arbeit möchten wir speziell jedes Element in zwei gleich große feinere Elemente teilen.
Eine weitere Möglichkeit ist die sogenannte *adaptive* Netzverfeinerung. Dabei werden nicht alle, sondern nur bestimmte Elemente verfeinert. Wir möchten insbesondere jene Elemente auswählen, für welche der lokale Fehleranteil am größten ist, um so den Fehler optimal zu minimieren. Dieses Auswahlkriterium realisieren wir mittels der *Dörfler Markierungsstrategie*. Dabei wählen wir für eine fest gewählte Konstante $\theta \in (0,1)$ eine bezüglich der Elementanzahl minimale Teilmenge $\mathcal{M}_h \subseteq \mathcal{T}_h$, sodass

$$\theta \sum_{T\in\mathcal{T}_h} \rho_h(T)^2 \leq \sum_{T\in\mathcal{M}_h} \rho_h(T)^2 \tag{3.32}$$

gilt. Es bezeichnen $\rho_h(T)$ die lokalen Anteile des Fehlerschätzers aus (3.30) beziehungsweise (3.31). Tatsächlich erreichen wir mit uniformer Netzverfeinerung, dass der Fehler schneller gegen 0 geht. Der Fehler konvergiert bei uniformer Netzverfeinerung in Abhängigkeit der Anzahl der Randelemente N mit einer Rate $\mathcal{O}(N^{-r})$.
Bei adaptiver Netzverfeinerung hängt die Rate von der Wahl der gewählten Diskretisierungsräume ab. Falls wir den Raum $H^{1/2}(\Gamma)$ mit $\mathcal{S}^1(\mathcal{T}_h)$ und den Raum $H^{-1/2}(\Gamma)$ mit $\mathcal{P}^0(\mathcal{T}_h)$ diskretisieren, erhalten wir eine optimale Konvergenzrate von $\mathcal{O}(N^{-3/2})$.
Für den Fall, dass wir $H^{1/2}(\Gamma)$ durch $\mathcal{S}^2(\mathcal{T}_h)$ und $H^{-1/2}(\Gamma)$ durch $\mathcal{P}^1(\mathcal{T}_h)$ ersetzten, erhalten wir eine noch bessere optimale Konvergenzrate $\mathcal{O}(N^{-5/2})$.

3.1.6 Lösungsalgorithmus

Wir möchten an dieser Stelle einen Algorithmus zur Berechnung von Φ_h auf einem Netz $\mathcal{T}_h$ angeben. Die Eingabeparameter des Algorithmus sind die Triangulierung $\mathcal{T}_h$ und die Funktion für die Dirichlet-Daten am Rand g. Der Rückgabewert ist der Koeffizientenvektor $\boldsymbol{x}_h$ von Φ_h bezüglich der gewählten Basis $\{\chi_1^0, \chi_2^0, \dots, \chi_N^0, \chi_1^1, \chi_2^1, \dots, \chi_N^1\}$ von $\mathcal{P}^1(\mathcal{T}_h)$. Für die Randelementmethode beziehungsweise für die indirekte Randintegralmethode sieht der Algorithmus folgendermaßen aus:

1. Berechne die rechte Seite:
   (a) Randelementmethode: Erzeuge die Matrix $\mathbf{K}$ und die Matrix $\mathbf{M}$ bezüglich der Triangulierung $\mathcal{T}_h$. Erzeuge den Koeffizientenvektor $\boldsymbol{g}_h$ der diskreten Funktion G_h bezüglich der Basis $\{\zeta_1^1, \zeta_2^1, \dots, \zeta_{\widetilde{N}}^1, \zeta_1^2, \zeta_2^2, \dots, \zeta_N^2\}$ von $\mathcal{S}^1(\mathcal{T}_h)$. Berechne $\mathbf{K}\boldsymbol{g}_h + 1/2\mathbf{M}\boldsymbol{g}_h$.
   (b) indirekte Randintegralmethode: Erzeuge den Koeffizientenvektor $\boldsymbol{g}_h$ der diskreten rechten Seite G_h bezüglich der Basis $\{\zeta_1^1, \zeta_2^1, \dots, \zeta_{\widetilde{N}}^1, \zeta_1^2, \zeta_2^2, \dots, \zeta_N^2\}$ von $\mathcal{S}^1(\mathcal{T}_h)$.
2. Erzeuge die Matrix $\mathbf{V}$ bezüglich der Triangulierung $\mathcal{T}_h$.
3. Löse das lineare Gleichungssystem
   (a) Randelementmethode:
   $$\mathbf{V}\boldsymbol{x}_h = \mathbf{K}\boldsymbol{g}_h + 1/2\mathbf{M}\boldsymbol{g}_h$$
   (b) indirekte Randintegralmethode:
   $$\mathbf{V}\boldsymbol{x}_h = \boldsymbol{g}_h$$
4. Gib $\boldsymbol{x}_h$ zurück.

3.1.7 Adaptiver Algorithmus

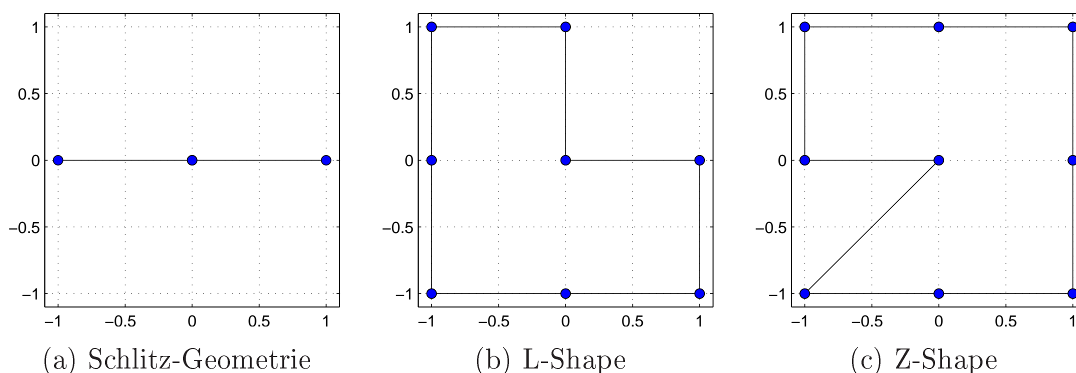
Für die Methode der adaptiven Netzverfeinerung möchten wir hier einen adaptiven Algorithmus angeben, der in jedem Schritt das Netz adaptiv verfeinert. Als Eingabeparameter erhält der Algorithmus das Startnetz $\mathcal{T}_0$ und die Funktion g. Außerdem erhält der Algorithmus ein fest gegebenes $0 < \theta < 1$ für die Dörfler Markierungsstrategie. Für diese Arbeit möchten wir speziell auch eine maximale Elementanzahl übergeben, damit der Algorithmus abbricht, sobald durch Verfeinern diese Anzahl erreicht wurde. Im Algorithmus

berechnen wir auch den Residualschätzer $\rho_h = \left(\sum_{T\in\mathcal{T}_h} \rho_h(T)^2\right)^{1/2}$ aus Abschnitt 3.1.4. Die lokalen Elementbeiträge $\rho_h(T)$ geben dabei eine Abschätzung für den Fehler zwischen der exakten Lösung ϕ_h mit gestörter rechter Seite und der Galerkin-Lösung Φ_h auf dem Element $T \in \mathcal{T}_h$ an. In Listenform lautet der Algorithmus folgendermaßen:

1. Berechne $\boldsymbol{x}_h$ laut Lösungsalgorithmus aus Abschnitt 3.1.6.
2. Berechne die Verfeinerungsindikatoren ρ_h für alle $T \in \mathcal{T}_h$.
3. Finde nun mit der Dörfler Markierungsstartegie eine bezüglich der Kardinalität minimale Teilmenge $\mathcal{M}_h \subseteq \mathcal{T}_h$.
4. Verfeinere zumindest alle Elemente $T \in \mathcal{M}_h$ und erhalte ein Netz $\widehat{\mathcal{T}}_h$.
5. Abbruch, falls die Elementzahl der Triangulierung $\widehat{\mathcal{T}}_h$ die vorgegebene maximale Elementzahl überschritten hat.
6. Setze nun $\mathcal{T}_h := \widehat{\mathcal{T}}_h$ und gehe zum ersten Schritt.

3.2 Numerische Beispiele

In diesem Abschnitt möchten wir einige numerische Beispiele vorstellen. Dabei möchten wir ein besonderes Augenmerk auf die unterschiedlichen Berechnungen der approximierenden Matrix $\mathbf{V}_p$ von $\mathbf{V}$ und $\mathbf{K}_p$ von $\mathbf{K}$ legen. Laut Abschnitt 2.4 gibt es zwei verschiedene Varianten die approximierende Matrix zu berechnen: einmal mit max-min-Kriterium und einmal mit min-Kriterium. In den folgenden Beispielen werden wir sehen, dass wir mit der Berechnung durch das max-min-Kriterium für den Ansatzraum $\mathcal{S}^2(\mathcal{T}_h)$ und den Testraum $\mathcal{P}^1(\mathcal{T}_h)$ ein besseres Genauigkeitsniveau erreichen.



(a) Schlitz-Geometrie (b) L-Shape (c) Z-Shape

Abbildung 1: Unterschiedliche Geometrien

Für die Berechnungen betrachten wir drei unterschiedliche Geometrien, darunter die Schlitz-Geometrie (Abbildung 1a), das L-Shape (Abbildung 1b) und das Z-Shape (Abbildung 1c). Diese drei Geometrien haben wir gewählt, weil die exakte Lösung für die betrachteten Beispiele Singularitäten am Rand beziehungsweise an den Ecken aufweist.

Für die Berechnungen haben wir ein vorgegebenes Startnetz verwendet und dieses dann adaptiv verfeinert. Dabei haben wir die Dörfler Markierungsstrategie (vgl. (3.32)) benützt mit $\theta = 0.5$ und dem gewichteten Residualschätzer ρ_h aus (3.31). Für die Berechnung

haben wir uns dann eine maximale Elementzahl vorgegeben, bis zu welcher gerechnet werden soll. Um schon im Vorhinein Instabilitäten geeignet abzufangen und somit eine Stagnation des Verfahrens rechtzeitig zu erkennen, haben wir in unserer Berechnung noch eine zusätzliche Abbruchbedingung eingeführt. Diese lautet:

Falls der berechnete Wert für ρ_h auf dem aktuellen Gitter größer ist als 10 mal das Minimum über alle bisher berechneten Werte für ρ_h, dann brich die Berechnung ab. (3.33)

3.2.1 Indirekte Randintegralmethode auf dem Schlitz

Wir betrachten die Laplace-Gleichung mit gegebenen Dirichlet-Randdaten

$$\begin{aligned} -\Delta u &= 0 \quad \text{auf } \Omega \subset \mathbb{R}^2, \\ u &= 1 \quad \text{auf } \Gamma := \partial\Omega, \end{aligned} \tag{3.34}$$

mit $\Omega := ([-1,1]\times\{0\})^C \subset \mathbb{R}^2$. Um (3.34) zu lösen, möchten wir hier mittels der indirekten Randintegralmethode vorgehen. Wir möchten die exakte Lösung ϕ von

$$V\phi = 1$$

für $1 \in H^{1/2}(\Gamma)$ unter Anwendung des Galerkin-Verfahrens approximieren. Dabei sei die Diskretisierung für Γ durch folgende Gitterpunkte gegeben:

$$\left\{ \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

Dieses Startnetz ist in Abbildung 1a dargestellt.
In Abbildung 2 betrachten wir den Fehler der approximierten Lösung Φ_h gegenüber der exakten Lösung ϕ in der Energienorm. Genauer gesagt betrachten wir den Fehler $\|\phi - \Phi_h\|$, wobei wegen der Galerkin-Orthogonallität gilt

$$\|\phi - \Phi_h\|^2 = \|\phi\| - \|\Phi_h\|^2.$$

Da die exakte Lösung und daher auch ihre Norm nicht bekannt sind, müssen wir $\|\phi\|$ geeignet approximieren. In jedem adaptiven Schritt berechnen wir dann $\|\Phi_h\|$. Diese Werte können wir nun mit dem Aitken'schen Deltaquadrat-Verfahren extrapolieren. Das Aitken'sche Deltaquadrat-Verfahren ist ein Verfahren zur Verbesserung der Konvergenzgeschwindigkeit von Folgen. Für eine Folge $(x_n)_{n\in\mathbb{N}}$ ist es definiert durch

$$y_n := x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n}.$$

Laut [6, Kapitel 5.2] wird so die Konvergenz der Folge $(y_n)_{n\in\mathbb{N}}$ beschleunigt. Wir wenden dieses Verfahren auf $\|\Phi_h\|$ an und erhalten mit dem letzten berechneten Wert, welcher der Approximation auf dem feinsten Gitter entspricht, eine gute Annäherung für den exakten Wert $\|\phi\|$.
In Abbildung 2 ist auf der x-Achse die Anzahl der Gitternetzkanten N aufgetragen. Für dieses Beispiel haben wir bis zu einer maximalen Elementanzahl $N = 1000$ gerechnet. Damit die Konvergenzraten des Fehlers optimal erkennbar sind, haben wir beide Koordinatenachsen logarithmisch skaliert. Wir sehen, dass in beiden Fällen die Fehler mit

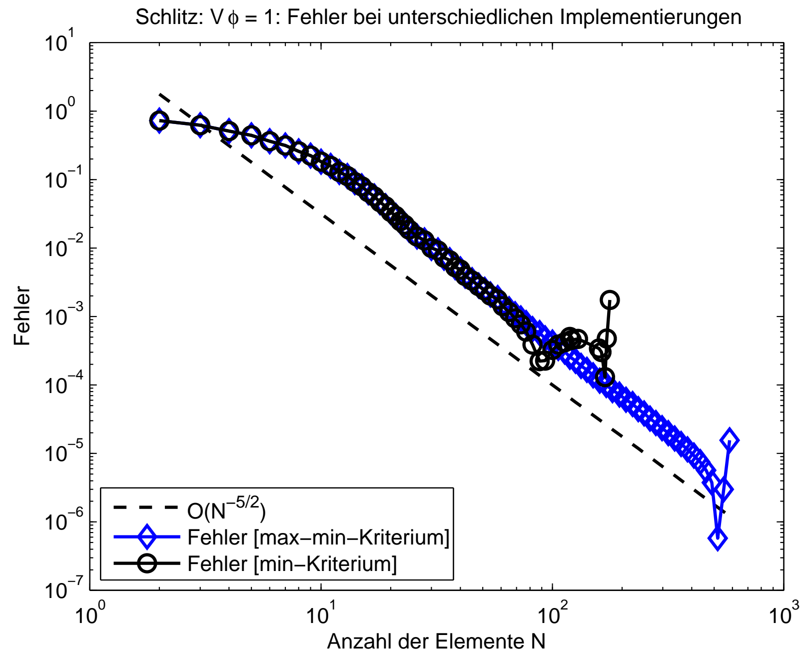


Abbildung 2: Schlitz: Fehler für Implementierung mit max-min-Kriterium vs. min-Kriterium

einer Rate $\mathcal{O}(N^{-5/2})$ gegen 0 streben. Allerdings sehen wir, dass bei der Implementierung mit min-Kriterium die Berechnung schon wesentlich früher instabil wird als mit max-min-Kriterium. Für das min-Kriterium erreichen wir nur ein Fehlerniveau von ungefähr $10^{-3.8}$, wohingegen wir mit dem max-min-Kriterium wesentlich genauer approximieren können und sogar bis $10^{-5.7}$ kommen. Obwohl wir eigentlich als maximale Elementzahl $N = 1000$ angegeben haben, wurde die Berechnung sowohl für das min-Kriterium als auch das max-min-Kriterium schon vorzeitig durch die Abbruchbedingung (3.33) abgebrochen. Bei der Implementierung mittels min-Kriterium begann das Verfahren bereits bei einer Elementanzahl von $N = 177$ zu stagnieren, wohingegen das Verfahren bei Implementierung mittels max-min-Kriterium erst bei einer Elementanzahl $N = 584$ zu stagnieren begann.
Im Vergleich sind in Abbildung 3 zusätzlich die Fehlerschätzer ρ_h dargestellt. Da auch hier beide Achsen logarithmisch skaliert sind, sehen wir, dass sich auch die Fehlerschätzer wie $\mathcal{O}(N^{-5/2})$ verhalten. Insbesondere beobachten wir hier die Effizienz und Zuverlässigkeit des gewichteten Residualschätzers. Für die Implementierung mittels min-Kriterium sehen wir auch, dass der Fehlerschätzer etwas später als der tatsächliche Fehler wieder zu wachsen beginnt. Im Falle der Implementierung mittels max-min-Kriterium beginnt der Schätzer ebenfalls etwas später als der Fehler zu springen. Bevor der Fehler jedoch tatsächlich wieder wächst, fällt er zuerst noch einmal stark. Das liegt vermutlich daran, dass wir für $|||\phi|||$ einen approximativen Wert der durch Extrapolation berechnet wurde, verwendet haben. Interessant ist auch, dass bei dem zum max-min-Kriterium gehörigen Residualschätzer ρ_h keine Sprünge sichtbar sind, obwohl die Abbruchbedingung bei einer Elementanzahl $N = 584$ in Kraft tritt. Tatsächlich ist es so, dass der Fehlerschätzer bei der Elementanzahl, bei welcher die Abbruchbedingung erfüllt wird, schon den Wert `Inf` annimmt, das entspricht in MATLAB dem Wert ∞.
Im Anschluss haben wir uns genauer angesehen, wie stark adaptiert das Gitter bereits ist.

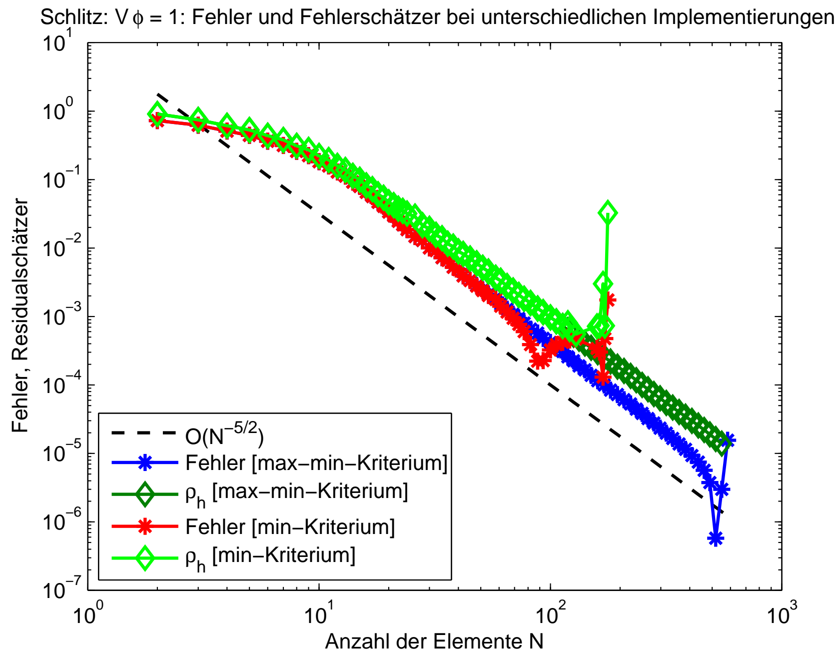


Abbildung 3: Schlitz: Fehler und Fehlerschätzer für Implementierung mit max-min-Kriterium vs. min-Kriterium

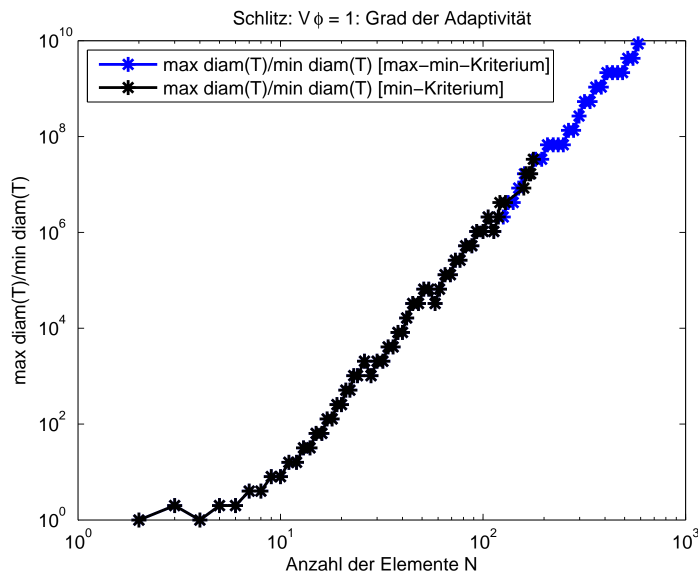


Abbildung 4: Schlitz: Grad der Adaptivität für Implementierung mit max-min-Kriterium vs. min-Kriterium

Für das Netz $\mathcal{T}_h$ sei der Grad der Adaptivität τ folgendermaßen definiert

$$\tau := \frac{\max_{T\in\mathcal{T}_h} \operatorname{diam}(T)}{\min_{T\in\mathcal{T}_h} \operatorname{diam}(T)}. \tag{3.35}$$

Dieser gibt für das aktuelle Netz $\mathcal{T}_h$ an, wie groß der Durchmesser des größten Elements im Vergleich zum kleinsten Element ist. In Abbildung 4 ist der Grad der Adaptivität τ für die

Implementierung mit min-Kriterium in schwarz dargestellt. Im Vergleich dazu ist in Blau τ für das max-min-Kriterium dargestellt. Wir sehen, dass das Netz bei der Implementierung mittels max-min-Kriterium noch weiter adaptiert wird. Allerdings liegt das daran, dass beim min-Kriterium die Berechnung schon früher aufgrund der Abbruchbedingung abgebrochen wird und daher auch nicht weiter verfeinert wird. Des Weiteren sehen wir auch, dass die Art der Implementierung des V-Operators keinen wesentlichen Unterschied in der Verfeinerung des Netzes auslöst. Für die Implementierung mittels max-min-Kriterium erreichen wir einen Grad der Adaptivität $\tau \approx 10^{10}$ für eine Elementanzahl von $N = 584$. Da das Netz an dieser Stelle nun schon sehr stark adaptiert ist, ist es gut möglich, dass die auftretenden Instabilitäten, die in Abbildung 2 und 3 für das max-min-Kriterium sichtbar sind, darin begründet liegen.
Tatsächlich haben wir bei der Berechnung festgestellt, dass bereits Elemente vorkommen, deren Durchmesser kleiner als 10^{-12} ist. Das entspricht nun schon fast der Rechengenauigkeit. Da bei der Integralberechnung durch die Transformation mit dem Durchmesser der Elemente multipliziert wird, ist es naheliegend, dass so die Werte der Integrale verfälscht werden. Diese Tatsache wäre dann auch auch für die Instabilitäten verantwortlich.

3.2.2 Indirekte Randintegralmethode auf dem L-Shape

Auch für das L-Shape betrachten die Laplace-Gleichung mit gegebenen Dirichlet-Randdaten wie in (3.34). Dabei ist $\Omega := (-1,1) \times (-1,1) \backslash [0,1] \times [0,1] \subset \mathbb{R}^2$. Auch hier möchten wir über die indirekte Randintegralmethode mittels dem Galerkin-Verfahren die exakte Lösung ϕ von $V\phi = 1 \in H^{1/2}(\Gamma)$ approximieren. Als Diskretisierung sei das Startnetz aus Abbildung 1b durch folgende Gitterpunkte gegeben:

$$\left\{ \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

Da auch hier die exakte Lösung nicht bekannt ist, werden wir den Fehler approximieren und für $|||\phi|||$ den extrapolierten Wert von $|||\Phi_h|||$ auf dem feinsten Gitter verwenden.
In Abbildung 5 ist also der Fehler $|||\phi - \Phi_h|||$ über der Anzahl der Elemente N aufgetragen. Wieder sind beide Achsen logarithmisch skaliert. Wir haben hier bis zu einer maximalen Elementanzahl von $N = 9000$ Elementen gerechnet. Auch hier sehen wir, dass die Fehler in beiden Fällen mit einer Rate $\mathcal{O}(N^{-5/2})$ gegen 0 streben. Obwohl für das max-min-Kriterium die Berechnung für bis zu 9000 Elementen größtenteils stabil verläuft, beginnt der Fehler für das min-Kriterium ab ungefähr 1000 Elementen wieder zu wachsen und das Verfahren stagniert. Bei 1024 Elementen tritt dann die Abbruchbedingung (3.33) in Kraft und die Berechnung bricht ab. Bei der Berechnung mittels min-Kriterium erreichen wir nur ein Fehlerniveau von ungefähr $10^{-3.8}$. Im Vergleich dazu erreichen wir bei einer Implementierung mittels max-min-Kriterium ein Genauigkeitsniveau von ungefähr 10^{-6} bei einer maximalen Elementanzahl von $N = 9000$. In Abbildung 5 sehen wir auch für den Fehler beim min-Kriterium eine Abrundung gegen Ende der Berechnung. Diese Abrundung ist vermutlich auf die Approximation von $|||\phi|||$ zurückzuführen. Dadurch, dass wir anstelle von $|||\phi|||$ den extrapolierten Wert der letzten Berechnung von $|||\Phi_h|||$ verwenden, bleibt noch ein gewisser Fehler übrig, der durch die Approximation zustande gekommen ist. Bei der Berechnung mit dem max-min-Kriterium treten ab einer Elementanzahl von ungefähr $N = 4000$ leichte Instabilitäten auf. Der Fehler verbessert sich nicht mehr, aber es ist auch keine wesentliche Verschlechterung sichtbar.
Betrachten wir Abbildung 6, so können wir sehen, dass diese Abrundung bei dem min-Kriterium entsprechenden Fehlerschätzer nicht zustande kommt. Die Fehlerschätzer ver-

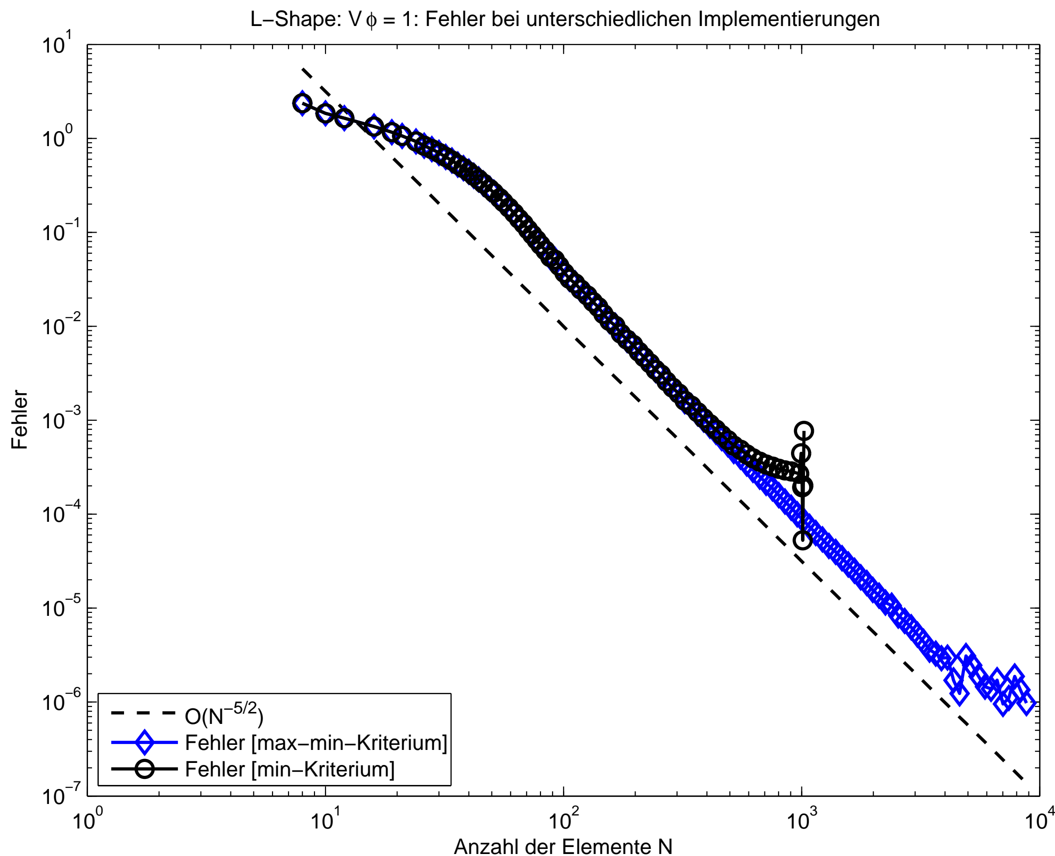


Abbildung 5: L-Shape: Fehler für Implementierung mit max-min-Kriterium vs. min-Kriterium

halten sich ähnlich wie die Fehler und streben ebenfalls mit einer Rate von $\mathcal{O}(N^{-5/2})$ gegen 0. Wir können auch feststellen, dass der Fehlerschätzer für die Implementierung mittels max-min-Kriterium im Vergleich zum Fehler keine Instabilitäten aufweist.
Auch für das L-Shape haben wir uns in Abbildung 7 den Grad der Adaptivität des Gitters τ aus (3.35) angesehen. Bei einer Elementanzahl knapp unter 9000 Elementen ist das Gitter für die Implementierung des V-Operators mittels max-min-Kriterium mit $\tau \approx 10^{10}$ bereits sehr stark adaptiert. Bei der Implementierung mittels min-Kriterium wurde bei 1024 Elementen abgebrochen, daher wurde dort auch das Gitter nicht mehr weiter verfeinert. Dennoch sehen wir kurz vor Abbruch eine deutliche Steigerung in der Adaptivität. Das heißt, es wurden gerade Elemente die ohnehin schon sehr klein waren noch weiter verfeinert. Dort muss also der Fehlerschätzer ρ_h, der bei uns als Indikator für die Verfeinerung dient, groß gewesen sein. Dies ist zusätzlich ein gutes Indiz für die Instabilität des Verfahrens bei der Implementierung mittels min-Kriterium.

3.2.3 Indirekte Randintegralmethode auf der Z-Shape

Für das Zshape betrachten wir ebenfalls die Laplace-Gleichung mit gegebenen Dirichlet-Randbedingungen wie in (3.34). Dabei ist

$$\Omega := (-1,1) \times (-1,1) \backslash \mathrm{conv} \left\{ \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ -1 \end{pmatrix} \right\} \subset \mathbb{R}^2,$$

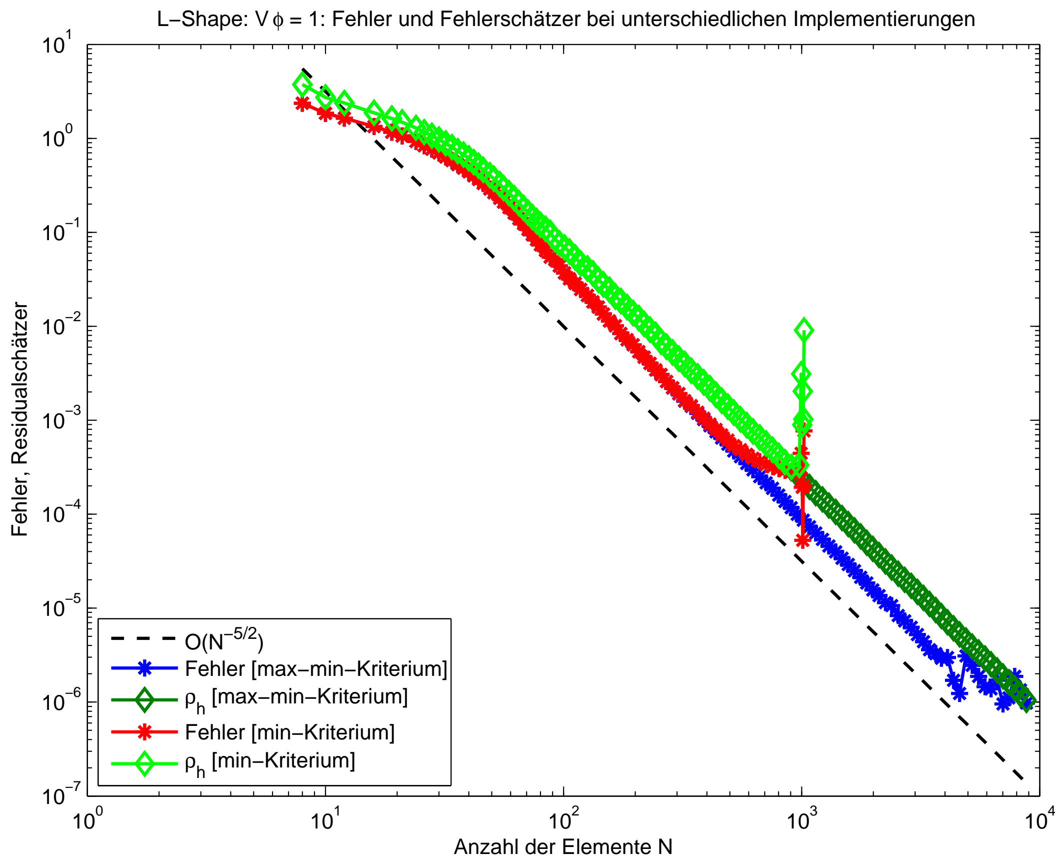


Abbildung 6: L-Shape: Fehler und Fehlerschätzer für Implementierung mit max-min-Kriterium vs. min-Kriterium

wobei conv$\{\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}\}$ die konvexe Hülle dreier Punkte $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^2$ bezeichnet.
Wie bei den anderen Geometrien auch, werden wir hier über die indirekte Randintegralmethode mittels dem Galerkin-Verfahren die exakte Lösung $\phi \in H^{-1/2}(\Gamma)$ von $V\phi = 1 \in H^{1/2}(\Gamma)$ approximieren. Um Γ zu diskretisieren haben wir das Startnetz (siehe Abbildung 1c) durch folgende Gitterpunkte vorgegeben:

$$\left\{ \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}.$$

Zuerst betrachten wir das Verhalten des Fehlers $\|\phi - \Phi_h\|$ bei adaptiver Netzverfeinerung. Auch verwenden wir als Approximation für $\|\phi\|$ den extrapolierten Wert von $\|\Phi_h\|$ auf dem feinsten Gitter.
In Abbildung 8 ist der Fehler $\|\phi - \Phi_h\|$ über der Anzahl der Elemente aufgetragen. Für die Berechnung haben wir uns eine maximale Elementanzahl $N = 5000$ vorgegeben. Da wir wieder beide Achsen logarithmisch skaliert haben, können wir für beide Fehler eine Konvergenzrate $\mathcal{O}(N^{-5/2})$ erkennen. Auch hier sehen wir, dass für die Implementierung des V-Operators mit min-Kriterium die Berechnung wesentlich früher instabil wird als mit max-min-Kriterium. Für das min-Kriterium erreichen wir bei einer Elementanzahl von $N = 1002$ ein Fehlerniveau von ungefähr $10^{-3.9}$. Dann tritt die Abbruchbedingung (3.33) in Kraft und die Berechnung wird abgebrochen. Gegen Ende hin können wir allerdings eine leichte Abflachung des Fehlers feststellen, welche vermutlich durch die Extrapolation zustande gekommen ist. Für das max-min-Kriterium sehen wir, dass die Berechnung länger

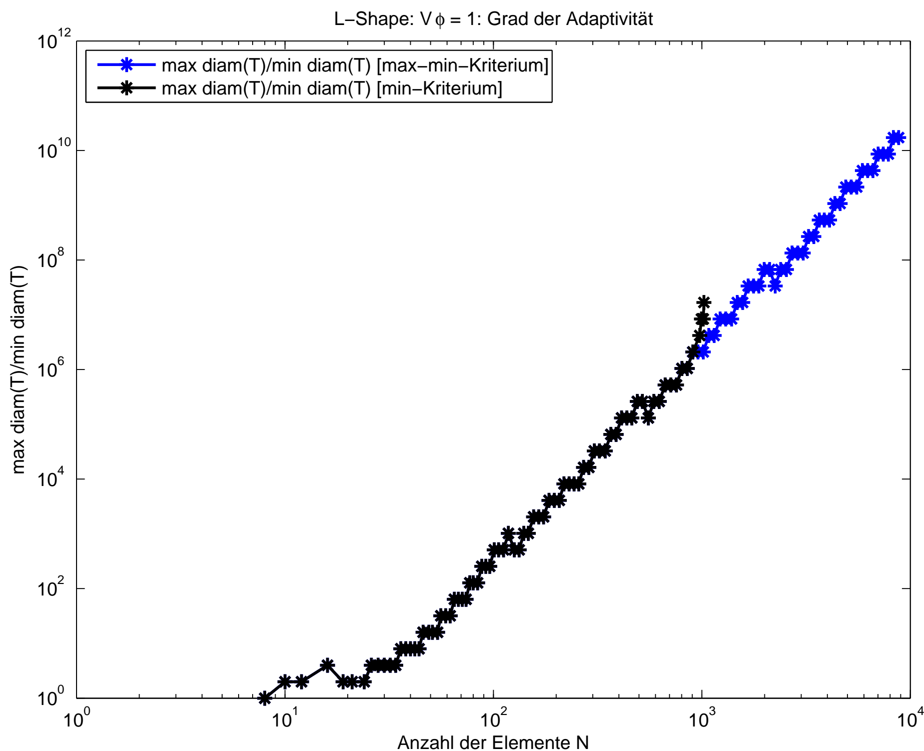


Abbildung 7: L-Shape: Grad der Adaptivität für Implementierung mit max-min-Kriterium vs. min-Kriterium

stabil läuft, bis zu einer Elementzahl von $N = 3897$ und einem Genauigkeitsniveau von $10^{-5.8}$. Dann setzt auch hier die Abbruchbedingung ein. Hier sehen wir keine Abrundung beim Fehler, die Berechnung wird gleich instabil.
In Abbildung 9 betrachten wir zusätzlich die Residualschätzer für die unterschiedlichen Implementierungen. Auch hier beobachten wir Effizienz und Zuverlässigkeit des Schätzers. Sowohl der Schätzer als auch der Fehler streben mit Rate $\mathcal{O}(N^{-5/2})$ gegen 0. Interessanterweise sind beim zum max-min-Kriterium gehörigen Residualschätzer ρ_h keine Sprünge sichtbar, obwohl die Abbruchbedingung (3.33) bei einer Elementanzahl $N = 3897$ in Kraft tritt. Tatsächlich ist es so, dass der Fehlerschätzer bei der Elementanzahl, bei welcher die Abbruchbedingung ausgelöst wird, schon den Wert `Inf` annimmt, das entspricht in MATLAB dem Wert ∞.
In Abbildung 10 sehen wir, dass das Netz für die beim max-min-Kriterium erreichte maximale Elementanzahl $N = 3897$ schon sehr stark adaptiert ist. Wir sehen, dass sich die unterschiedliche Implementierung des V-Operators bis zu einer gewissen Elementanzahl nicht auf die Adaptivität des Netzes auswirkt. Für das min-Kriterium wurde die Berechnung allerdings bei $N = 1002$ abgebrochen und daher wurde auch nicht weiter verfeinert. Bevor die Berechnung abgebrochen wurde, sehen wir, dass der Grad der Adaptivität nochmals etwas stärker ansteigt. Dies ist ein Indiz dafür, dass Elemente, die ohnehin schon sehr klein sind, noch weiter verfeinert werden, da dort der Fehlerschätzer ρ_h am größten ist.
Für die Implementierung mittels max-min-Kriterium erreichen wir einen Grad der Adaptivität von ungefähr $\tau \approx 10^{10}$ für eine Elementanzahl von $N = 3897$. Tatsächlich haben wir bei der Berechnung festgestellt, dass bereits Elemente vorkommen, deren Durchmesser

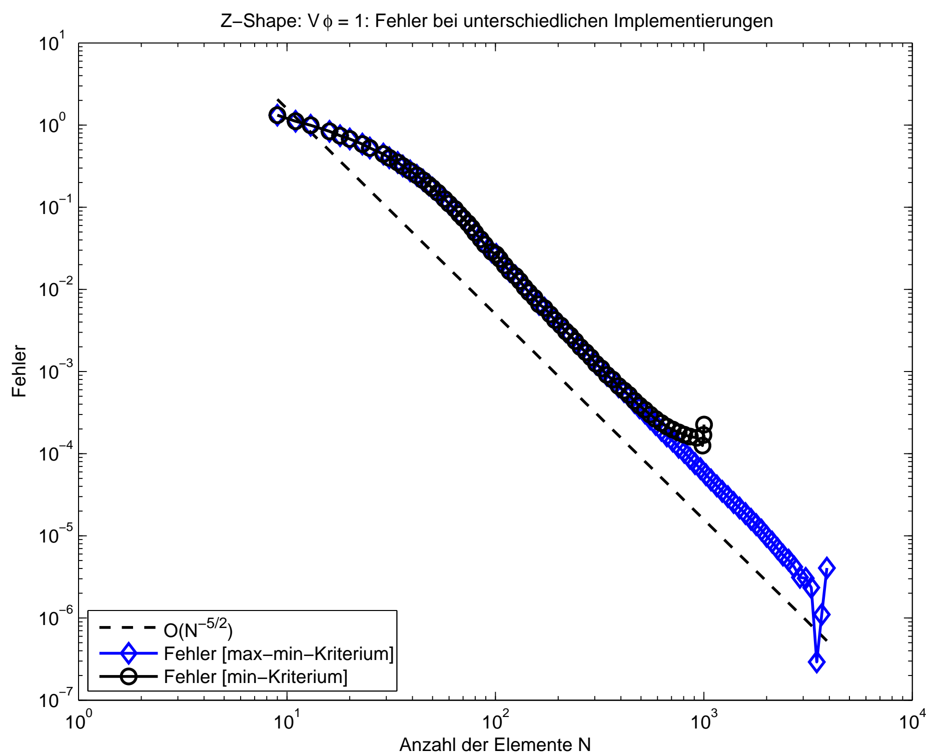


Abbildung 8: Z-Shape: Fehler für Implementierung mit max-min-Kriterium vs. min-Kriterium

kleiner als 10^{-12} ist, was beinahe der Maschinengenauigkeit entspricht. Da bei der Integralberechnung durch die Transformation mit dem Durchmesser der Elemente multipliziert wird, ist es gut möglich, dass die Berechnung so verfälscht wird. Möglicherweise ist das auch Grund dafür, dass der Fehlerschätzer ρ_h für das max-min-Kriterium den Wert ∞ erreicht.

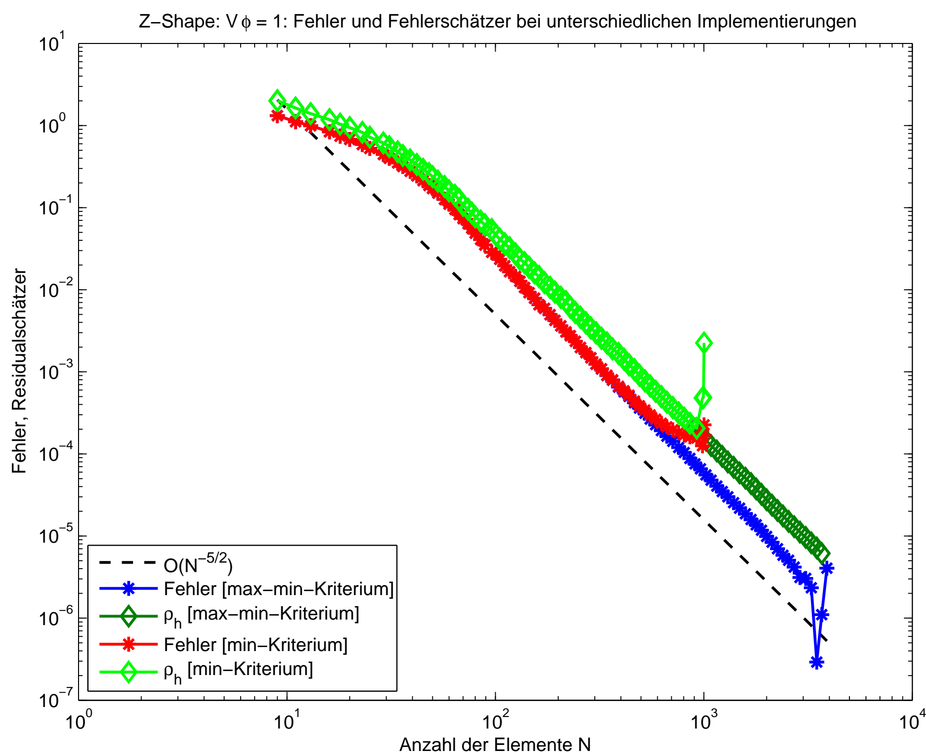


Abbildung 9: Z-Shape: Fehler und Fehlerschätzer für Implementierung mit max-min-Kriterium vs. min-Kriterium

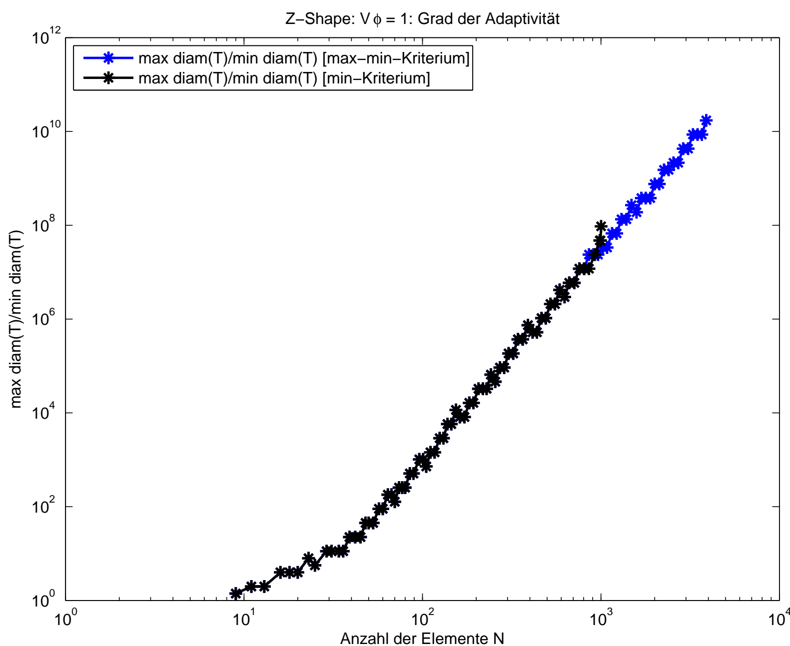


Abbildung 10: Z-Shape: Grad der Adaptivität für Implementierung mit max-min-Kriterium vs. min-Kriterium

4 Vergleich der Implementierungen

Abschließend möchten wir noch den Programmcode zur stabilen Berechnung des V-Operators und des K-Operators, der im Rahmen dieser Bachelorarbeit erweitert wurde, vorstellen.
Für den Ansatzraum $\mathcal{S}^1(\mathcal{T}_h)$ und den Testraum $\mathcal{P}^0(\mathcal{T}_h)$ gibt es bereits die MATLAB Programm-Bibliothek HILBERT für die adaptive Randelementmethode. HILBERT umfasst stabile Implementierungen des Einfach- und Doppelschichtpotential-Operators und des hypersingulären Integral-Operators für die Poisson- und Laplace-Gleichung in 2D für die $\mathcal{S}^1(\mathcal{T}_h)$-$\mathcal{P}^0(\mathcal{T}_h)$-Variante. Es stehen auch verschiedene Fehlerschätzer, Markierungsstrategien und Visualisierungswerkzeuge zur Verfügung. Zur Demonstration enthält HILBERT auch einige Beispiele, für welche die adaptive Randelementmethode für das Dirichlet- und Neumann-Problem sowie das gemischte Problem (mit Dirichlet- und Neumann-Bedingungen) implementiert ist. Für akademische Zwecke steht HILBERT zum Download unter `http://www.asc.tuwien.ac.at/abem/hilbert/` zur Verfügung.
In den letzten Jahren wurde die Implementierung von HILBERT dahingehend erweitert, dass nun auch die adaptive Randelementmethode für die $\mathcal{S}^2(\mathcal{T}_h)$-$\mathcal{P}^1(\mathcal{T}_h)$-Variante implementiert ist. Die Implementierungen des Einfach- und Doppelschichtpotential-Operators, sowie die Berechnung der Fehlerschätzer, Oszillationsterme, Fehler, etc. wurden ergänzt. Das Problem dieser Implementierung war allerdings, dass diese noch nicht stabil lief. Es war noch nicht möglich das gleiche Genauigkeitsniveau des Approximationsfehlers wie bei der $\mathcal{S}^1(\mathcal{T}_h)$-$\mathcal{P}^0(\mathcal{T}_h)$-Variante mit adaptiver Netzverfeinerung zu erreichen. Ziel dieser Arbeit war nun ein besseres Genauigkeitsniveau des Approximationsfehlers der exakten Lösung unter Einhalten der vorgegebenen Konvergenzrate $\mathcal{O}(N^{-5/2})$ zu erreichen. Dies ist durch Verbesserung des Programmcodes zur Berechnung des Einfachschicht- und Doppelschichtpotential-Operators geschehen. Für die Implementierungen der Einfachschichtpotential-Operatormatrix $\mathbf{V}$ und der Doppelschichtpotential-Operatormatrix $\mathbf{K}$ ist es notwendig bestimmte Doppelintegrale (siehe (3.16) beziehungsweise (3.18)) zu berechnen. Im bisher vorhandenen Code wurde zur Approximation in den Funktionen `computeWij` (Abschnitt 4.1.4), `computeVP1P1ij` (Abschnitt 4.1.8), `computeKij` (Abschnitt 4.2.3) und `computeKP1S2ij` (Abschnitt 4.2.9) das min-Kriterium herangezogen. Diese Programmcodes wurden nun dahingehend verbessert, dass nun statt dem min-Kriterium, das max-min-Kriterium zur Berechnung der approximierenden Matrix verwendet wird. Dafür war es notwendig, zusätzliche Funktionen zu implementieren, um auch beide Doppelintegrale aus (siehe (3.16) beziehungsweise (3.18)) mittels Quadratur berechnen zu können. Im Rahmen dieser Arbeit wurden also die Implementierungen von $\mathbf{V}$ um die Funktionen `computeWijFullQuadrature` (Abschnitt 4.1.7) und `computeVP1P1ijFullQuadrature` (Abschnitt 4.1.11) und von $\mathbf{K}$ um die Funktionen `computeKijFullQuadrature` (Abschnitt 4.2.8) und `computeKP1S2ijFullQuadrature` (Abschnitt 4.2.12) erweitert.
Die Implementierungen von $\mathbf{V}$ und $\mathbf{K}$ möchten wir im folgenden genau erklären. Die Implementierung erfolgt in C über die C-Schnittstelle von MATLAB, da die zugehörigen Matrizen in C schneller aufgebaut werden können.
Davor möchten wir allerdings noch einige vordefinierte Konstanten genauer erklären, die in den Programmcodes verwendet werden. Darunter ist eine vordefinierte Zulässigkeitskonstante

$$\texttt{DEFAULT_ETA} := 0.5, \tag{4.1}$$

auf welche zugegriffen wird, falls η nicht vom Benutzer spezifiert wird. Für die Berechnung der Matrixeinträge von $\mathbf{V}$ und $\mathbf{K}$ wird auch die Konstante π gebraucht welche in C nicht

vordefiniert ist und daher durch

$$\texttt{M_PI} := 3.14159265358979323846 \tag{4.2}$$

approximiert wird. Für die Implementierung ist es auch immer wieder notwendig zu überprüfen ob gewisse Werte gleich 0 beziehungsweise sehr klein sind. Aufgrund der Gleitkommadarstellung der Zahlen überprüfen wir nicht ob diese Werte gleich 0 sind, sondern ob der Wert kleiner als eine gewisse Maschinengenauigkeitskonstante ist. Hierfür haben wir

$$\texttt{EPS} := 10^{-12} \tag{4.3}$$

definiert.
Des Weiteren ist auch eine vorgegebene Anzahl an Knoten für die Gauß-Quadratur festgelegt:

$$\texttt{GAUSS_ORDER} := 16. \tag{4.4}$$

Falls also ein Integral mittels Quadratur berechnet wird, wird immer die Gauß-Quadratur $Q_{\texttt{GAUSS_ORDER}-1}$ verwendet.
Darüber hinaus möchten wir noch einige verwendete Funktionen erwähnen, deren genaue Implementierung nicht angegeben wird. Für die η-Zulässigkeitsbedingungen wird für die Berechnung von $\operatorname{dist}(T_i, T_j)$ beispielsweise die Funktion `distanceSegmentToSegment` verwendet. Die Funktion `getGaussPoints(GAUSS_ORDER)` gibt die Knotenpunkte z_k für $k = 0, \dots, \texttt{GAUSS_ORDER} - 1$ für die Gauß-Quadratur der Länge `GAUSS_ORDER` in Form eines Arrays zurück. Die Funktion `getGaussWeights(GAUSS_ORDER)` gibt dann die zugehörigen Gewichte ω_k für $k = 0, \dots, \texttt{GAUSS_ORDER} - 1$ auch in Form eines Arrays zurück.

4.1 Implementierung des V-Operators

Die Schnittstelle zu MATLAB ist durch die Funktion `mexFunction`, welche die Eingabeparameter einliest und anschließend den Speicher für die Matrix $\mathbf{V}$ allokiert, gegeben. Über die Funktion `computeVP1` wird dann die Berechnung der einzelnen Einträge der Matrix realisiert. Die Einträge der Submatrix $\mathbf{V}^{(00)}$ (vgl. (3.15)) werden dann mit der Funktion `computeVij`, welche nun in verbesserter Weise unter Anwendung des max-min-Kriteriums das Doppelintegral

$$\begin{aligned} \mathbf{V}_{ij}^{(00)} &= -\frac{1}{2\pi} \int_{T_i} \int_{T_j} \log|\boldsymbol{x} - \boldsymbol{y}|\, ds_{\boldsymbol{y}} ds_{\boldsymbol{x}} \\ &= -\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{8\pi} \int_{-1}^{1} \int_{-1}^{1} \log|\gamma_i(s) - \gamma_j(t)|\, dt\; ds \end{aligned} \tag{4.5}$$

berechnet. Da die Berechnung der Matrix $\mathbf{W}$ des hypersingulären Integraloperators W der Berechnung von $\mathbf{V}$ sehr ähnlich ist, greift die Funktion `computeVij` auf die Funktion `computeWij` zu. In dieser Funktion ist dann tatsächlich die Fallunterscheidung für das max-min-Kriterium implementiert. Falls die Elemente T_i, T_j η-unzulässig sind oder die Zulässigkeitskonstante η gleich 0 ist, so wird `computeWijAnalytic` aufgerufen. Für den Fall, dass die Elemente η-min-zulässig sind, wird die Funktion `computeWijSemianalytic` aufgerufen. Falls T_i, T_j η-max-zulässig sind, wird das Doppelintegral (4.5) mit der Funktion `computeWijFullQuadrature` berechnet. Die Funktionen `slp` und `slpIterative` berechnen dann Integrale der Form

$$\int_{-1}^{1} s^k \log|s\boldsymbol{u} + \boldsymbol{v}|^2\, ds \quad \text{für } k \in \mathbb{N}_0 \text{ und } \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^2$$

mit Hilfe derer dann konkret das geforderte Doppelintegral ausgerechnet werden kann.

Die anderen Submatrizen $\mathbf{V}^{(01)}, \mathbf{V}^{(10)}$ und $\mathbf{V}^{(11)}$ werden mit der Funktion `computeVP1P1ij` berechnet. Mit Hilfe des max-min-Kriteriums berechnet nun diese Funktion eine Approximation für das Doppelintegal

$$\begin{aligned} \mathbf{V}_{ij}^{(\ell m)} &= -\frac{1}{2\pi}\int_{T_i}\int_{T_j} \log\left(|\boldsymbol{x}-\boldsymbol{y}|\right)\chi_j^m(\boldsymbol{y})\chi_i^\ell(\boldsymbol{x})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \\ &= -\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\log\left(|\gamma_i(s)-\gamma_j(t)|\right)t^m s^\ell dt\, ds \end{aligned} \tag{4.6}$$

mit $\ell, m \in \{0,1\}$ und $(\ell,m) \neq (0,0)$. Falls die Randelemente T_i, T_j η-unzulässig sind oder die Zulässigkeitskonstante η gleich 0 ist, so wird die Funktion `computeVP1P1ijAnalytic` aufgerufen. Wenn die Elemente T_i, T_j η-min-zulässig sind, wird das Doppelintegral (4.6) mit der Funktion `computeVP1P1ijSemiAnalytic` berechnet. Falls die zwei Randelemente η-max-zulässig sind, so wird die Funktion `computeVp1P1ijFullQuadrature` aufgerufen. Zur analytischen Berechnung von (4.6) wird in weiterer Folge die Funktion `doubleSlp` herangezogen, welche

$$\int_{-1}^{1}\int_{-1}^{1} s^k t^\ell \log|s\boldsymbol{u}+t\boldsymbol{v}+\boldsymbol{w}|\,dt\,ds \quad \text{für } k,\ell\in\mathbb{N}_0 \text{ und } \boldsymbol{u},\boldsymbol{v},\boldsymbol{w}\in\mathbb{R}^2$$

berechnet.

4.1.1 `mexFunction`

Die Funktion erhält als Eingabeparameter die Arrays `coordinates` und `elements`, welche die Triangulierung $\mathcal{T}_h$ beschreiben. Für die $\widetilde{N}\times 2$-Matrix `coordinates` stehen in der j-ten Zeile die Koordinaten eines Randknotens $\boldsymbol{z}_j \in \mathbb{R}^2$. Für die $N \times 2$-Matrix `elements` beschreibt jede Zeile ein Randelement, wobei die Einträge von `elements` nicht die Koordinaten selbst, sondern lediglich die Indizes der zugehörigen Zeile in `coordinates`, enthalten. Optional kann auch noch die Zulässigkeitskonstante η übergeben werden. Falls dies nicht geschieht, wird für η der vordefinierte Wert `DEFAULT_ETA` verwendet.

Listing 1: mexFunction

```
28 void mexFunction(int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[]) {
29   const char* function_name = mexFunctionName();
30   char error_message[255];
31   int nc = 0, ne = 0;
32   double* V = NULL;
33   const double* elements;
34   const double* coordinates;
35   double eta = DEFAULT_ETA;
36
37   if (nlhs != 1) {
38     sprintf(error_message,
39             "Invalid number of output arguments. Use either\n"
40             "  V = %s(coordinates,elements)\n"
41             "or\n"
42             "  V = %s(coordinates,elements,eta)\n",
43             function_name,function_name);
44     mexErrMsgTxt(error_message);
45   }
46
47   /* Read input data */
48   switch (nrhs) {
49     case 3:
```

```
      eta = (double) mxGetScalar(prhs[2]);
    case 2:
      coordinates = (const double*) mxGetPr(prhs[0]);
      nc          = mxGetM(prhs[0]);
      elements    = (const double*) mxGetPr(prhs[1]);
      ne          = mxGetM(prhs[1]);
      break;
    default:
      sprintf(error_message,
              "Invalid␣number␣of␣input␣arguments␣(%d).␣Use␣either\n"
              "␣␣V␣=␣%s(coordinates,␣elements)\n"
              "or\n"
              "␣␣V␣=␣%s(coordinates,␣elements,␣eta)\n",
              nrhs, function_name, function_name);
      mexErrMsgTxt(error_message);
      return;
  }

  if (eta < 0) {
    sprintf(error_message,
      "In␣%s,␣ETA␣is␣less␣than␣zero␣(ETA=%lf).\n"
      "Please␣choose␣a␣value␣that␣is␣greater␣than␣or␣equal␣to␣0.\n",
      function_name, eta);
    mexErrMsgTxt(error_message);
  }

  /* Allocate output data */
  plhs[0] = mxCreateDoubleMatrix(2*ne,2*ne,mxREAL);
  V = mxGetPr(plhs[0]);

  computeVP1(V, nc, ne, coordinates, elements, eta);
}
```

4.1.2 computeVP1

Die Funktion erhält als Eingabeparameter die Arrays `coordinates`, `elements` und deren Dimension, also die Elementanzahl N und die Anzahl der Randknoten $\widetilde{N}$. Zusätzlich wird ein Pointer übergeben, der auf den für $\mathbf{V}$ allokierten Speicher zeigt. Auch die Zulässigkeitskonstante η wird übergeben. Nun durchläuft die Funktion alle Kombinationen von Elementen $T_i \times T_j$ und berechnet dazu den entsprechenden Eintrag der Matrix $\mathbf{V}$. Für die Berechnung der Submatrix $\mathbf{V}^{(00)}$ wird dabei die Funktion `computeVij` aufgerufen, für die anderen drei Submatrizen $\mathbf{V}^{(01)}, \mathbf{V}^{(10)}$ und $\mathbf{V}^{(11)}$ wird die Funktion `computeVP1P1ij` aufgerufen.

Listing 2: computeVP1

```
void computeVP1(double* V, int nc, int ne,
                const double* coordinates, const double* elements, double eta) {

  int    i = 0, j = 0;
  int    aidx = 0, bidx = 0, cidx = 0, didx = 0;
  double a0 = 0., a1 = 0., b0 = 0., b1 = 0.;
  double c0 = 0., c1 = 0., d0 = 0., d1 = 0.;

  assert(eta >= 0);
/*P0xP0, P1xP1 - Abschnitt */
  for (i=0; i<ne; ++i) {
    aidx = (int) elements[i]-1;        /* 1st node of element T_i = [A,B] */
    a0 = coordinates[aidx];
    a1 = coordinates[aidx+nc];

    bidx = (int) elements[i+ne]-1;     /* 2nd node of element T_i = [A,B] */
    b0 = coordinates[bidx];
    b1 = coordinates[bidx+nc];

    for (j=i; j<ne; ++j) {
      cidx = (int) elements[j]-1;      /* 1st node of element T_j = [C,D] */
      c0 = coordinates[cidx];
```

```
105        c1 = coordinates[cidx+nc];
106
107        didx = (int) elements[j+ne]-1;    /* 2nd node of element T_j = [C,D] */
108        d0 = coordinates[didx];
109        d1 = coordinates[didx+nc];
110
111        /* P0xP0*/
112        V[i + 2*j*ne] = computeVij(a0,a1,b0,b1, c0,c1,d0,d1,eta);
113        V[j + 2*i*ne] = V[i + 2*j*ne];
114
115        /*P1xP1*/
116        V[2*ne*ne + i + 2*j*ne+ne] =
117        computeVP1P1ij(a0,a1,b0,b1,c0,c1,d0,d1,1,1,eta);
118        V[2*ne*ne + j + 2*i*ne+ne] = V[2*ne*ne + i + 2*j*ne+ne];
119      }
120    }
121  /*P0xP1, P1xP0 - Abschnitt */
122    for (i=0; i<ne; ++i) {
123      aidx = (int) elements[i]-1;         /* 1st node of element T_i = [A,B] */
124      a0 = coordinates[aidx];
125      a1 = coordinates[aidx+nc];
126
127      bidx = (int) elements[i+ne]-1;      /* 2nd node of element T_i = [A,B] */
128      b0 = coordinates[bidx];
129      b1 = coordinates[bidx+nc];
130
131      for (j=0; j<ne; ++j) {
132        cidx = (int) elements[j]-1;       /* 1st node of element T_j = [C,D] */
133        c0 = coordinates[cidx];
134        c1 = coordinates[cidx+nc];
135
136        didx = (int) elements[j+ne]-1;    /* 2nd node of element T_j = [C,D] */
137        d0 = coordinates[didx];
138        d1 = coordinates[didx+nc];
139
140        /* P1xP0*/
141        V[ne + i + 2*j*ne] = computeVP1P1ij(a0,a1,b0,b1, c0,c1,d0,d1,1,0,eta);
142        /* P0xP1 */
143        V[2*ne*ne + j + 2*i*ne] = V[ne+i+2*j*ne];
144      }
145    }
146  }
```

4.1.3 computeVij

Für die Elemente $T_i = [\boldsymbol{a}, \boldsymbol{b}]$ und $T_j = [\boldsymbol{c}, \boldsymbol{d}]$ erhält die Funktion die Randpunkte $\boldsymbol{a} = [\texttt{a0}, \texttt{a1}]$, $\boldsymbol{b} = [\texttt{b0}, \texttt{b1}]$, $\boldsymbol{c} = [\texttt{c0}, \texttt{c1}]$ und $\boldsymbol{d} = [\texttt{d0}, \texttt{d1}]$. Zusätzlich wird die Zulässigkeitskonstante η übergeben. Die Funktion `computeVij` ist eigentlich nur für eine Skalierung mit $\text{diam}(T_i)\text{diam}(T_j)$ verantwortlich, die für eine Transformation $T_i \to [-1, 1]$ und $T_j \to [-1, 1]$ notwendig ist.

Listing 3: computeVij

```
328  double computeVij(double a0, double a1, double b0, double b1,
329        double c0, double c1, double d0, double d1, double eta) {
330    /*
331     * INPUT:  elements Ti = [a,b], Tj = [c,d] with a,b,c,d \in \R^2
332     * OUTPUT: Galerkin integral
333     *         -1/(2pi) * \int_{Tj} \int_{Ti} log|x-y| ds_y ds_x
334     */
335    double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1); /* hi = norm(b-a)^2 */
336    double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1); /* hj = norm(d-c)^2 */
337
338    return sqrt(hi*hj)*computeWij(a0, a1, b0, b1, c0, c1, d0, d1, eta);
339  }
```

4.1.4 `computeWij`

Die Funktion `computeWij` ist nun tatsächlich für die Realisierung des max-min-Kriteriums zuständig, welche in den Zeilen 368-376 stattfindet. Hier wurde das min-Kriterium durch das max-min-Kriterium ersetzt. Davor wird noch sichergestellt, dass T_i das längere Randstück ist. Dies ist für den Fall der semianalytischen Berechnung wichtig, da dort das Integral über das kürzere Randelement durch Quadratur ersetzt wird. Gegebenenfalls werden also T_i und T_j vertauscht.
Zurückgegeben wird eine Approximation für das Integral

$$-\frac{1}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\log|\gamma_i(s)-\gamma_j(t)|\,dt\,ds. \tag{4.7}$$

Listing 4: computeWij

```
341 double computeWij(double a0, double a1, double b0, double b1,
342       double c0, double c1, double d0, double d1, double eta) {
343   /*
344    * INPUT:  elements Ti = [a,b], Tj = [c,d] with a,b,c,d \in \R^2
345    * OUTPUT: Galerkin integral -1/(2pi) \int_{Tj} \int_{Ti} log|x-y| ds_y ds_x
346    */
347
348   double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1); /* hi = norm(b-a)^2 */
349   double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1); /* hj = norm(d-c)^2 */
350   double tmp = 0.;
351
352   /* For stability reasons, we guarantee   hj <= hi   to ensure that *
353    * outer integration is over smaller domain. This is done by       *
354    * swapping Tj and Ti if necessary.                                */
355
356   if (hj > hi) {
357     tmp = a0; a0 = c0; c0 = tmp;   /* swap a and c */
358     tmp = a1; a1 = c1; c1 = tmp;
359     tmp = b0; b0 = d0; d0 = tmp;   /* swap b and d */
360     tmp = b1; b1 = d1; d1 = tmp;
361     tmp = hi; hi = hj; hj = tmp;   /* ensure that hj <= hi */
362   }
363
364   if ( eta == 0) { /* compute all matrix entries analytically */
365     return computeWijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1);
366   }
367   else { /* compute admissible matrix entries semi-analytically */
368     if ( eta*distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1) > sqrt(hi) ) {
369       return computeWijFullQuadrature(a0,a1, b0,b1, c0,c1, d0,d1);
370     }
371     else if ( eta*distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1) > sqrt(hj) ) {
372       return computeWijSemianalytic(a0,a1, b0,b1, c0,c1, d0,d1);
373     }
374     else {
375       return computeWijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1);
376     }
377   }
378 }
```

4.1.5 `computeWijAnalytic`

Die Funktion berechnet das Integral (4.7) analytisch. Für Details zu dieser Berechnung möchten wir auf [10, Lemma 3.1] verweisen.

Listing 5: computeWijAnalytic

```
380 double computeWijAnalytic(double a0, double a1, double b0, double b1,
381                double c0, double c1, double d0, double d1) {
382   /*
383    * INPUT:  elements Ti = [a,b], Tj = [c,d] with a,b,c,d \in \R^2
```

```
384    * OUTPUT: Galerkin integral
385    *         -1/(2pi) * 1/|Ti| * 1/|Tj| \int_{Tj} \int_{Ti} log|x-y| ds_y ds_x
386    */
387
388   double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1); /* hi = norm(b-a)^2 */
389   double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1); /* hj = norm(d-c)^2 */
390   double val = 0., det = 0.;
391   double x[2], y[2], z[2];
392   double zxp[2], zxm[2], zyp[2], zym[2];
393   double lambda, mu;
394
395   x[0] = 0.5*(b0 - a0);             /* x = (b-a)/2 */
396   x[1] = 0.5*(b1 - a1);
397   y[0] = 0.5*(c0 - d0);             /* y = (c-d)/2 */
398   y[1] = 0.5*(c1 - d1);
399   z[0] = 0.5*(a0 + b0 - c0 - d0);   /* z = (a+b-c-d)/2 */
400   z[1] = 0.5*(a1 + b1 - c1 - d1);
401
402   zxp[0] = z[0] + x[0];             /* zxp = z+x = (2b-c-d)/2 */
403   zxp[1] = z[1] + x[1];
404   zxm[0] = z[0] - x[0];             /* zxm = z-x = (2a-c-d)/2 */
405   zxm[1] = z[1] - x[1];
406   zyp[0] = z[0] + y[0];             /* zyp = z+y = (a+b-2d)/2 */
407   zyp[1] = z[1] + y[1];
408   zym[0] = z[0] - y[0];             /* zym = z-y = (a+b-2c)/2 */
409   zym[1] = z[1] - y[1];
410
411   /* There hold different recursion formulae if Ti and Tj */
412   /* are parallel (det = 0) or not                          */
413
414   det = x[0]*y[1] - x[1]*y[0];
415
416   if ( fabs(det) <= EPS*sqrt(hi*hj) ) { /* case that x and y are linearly */
417     if ( fabs(x[0]) < fabs(x[1]) )      /* dependent, i.e., Ti and Tj are */
418       lambda = y[1] / x[1];             /* parallel. */
419     else
420       lambda = y[0] / x[0];
421
422     val = 0.5*( lambda * ( slp(1, y, zxm) - slp(1, y, zxp) )
423                         + slp(0, x, zyp) + slp(0, x, zym) );
424   }
425
426   else { /* case that x and y are linearly independent */
427     lambda = (z[0]*y[1] - z[1]*y[0]) /det;
428     mu = (x[0]*z[1] - x[1]*z[0]) /det;
429
430     val = 0.25 * ( -8 + (lambda+1)*slp(0, y, zxp) - (lambda-1)*slp(0, y, zxm)
431                        + (mu+1)*slp(0, x, zyp) - (mu-1)*slp(0, x, zym));
432   }
433
434   return -0.125*val /M_PI; /* = -1/(8*M_PI)*val */
435 }
```

4.1.6 computeWijSemianalytic

Die Funktion berechnet das Integral (4.7) semianalytisch, das heißt es wird

$$-\frac{1}{16\pi}\sum_{k=0}^{p}\omega_k\int_{-1}^{1}\log|\gamma_i(s)-\gamma_j(z_k)|^2\,ds$$

berechnet. Die Skalierung mit $1/2$ kommt durch $\log|\gamma_i(s)-\gamma_j(z_h)|^2 = 2\log|\gamma_i(s)-\gamma_j(z_h)|$ zustande und ist notwendig, damit das innere Integral mit der Funktion `slp` berechnet werden kann. Für unsere Berechnung ist $p = \texttt{GAUSS_ORDER} - 1$ aus (4.4).

Listing 6: computeWijSemianalytic

```
437 double computeWijSemianalytic(double a0, double a1, double b0, double b1,
438                               double c0, double c1, double d0, double d1) {
```

```
439   /*
440    * INPUT:  elements Ti = [a,b], Tj = [c,d] with a,b,c,d \in \R^2
441    * OUTPUT: Galerkin integral
442    *         -1/(2pi) * 1/|Ti| * 1/|Tj| \int_{Tj} \int_{Ti} log|x-y| ds_y ds_x
443    *         where outer integration is performed by Gaussian quadrature
444    */
445   int k;
446   double u[2], v[2];
447   double val = 0;
448   double sx0 = 0;
449   double sx1 = 0;
450   const double* gauss_point;
451   const double* gauss_wht;
452
453   gauss_point = getGaussPoints(GAUSS_ORDER);
454   gauss_wht   = getGaussWeights(GAUSS_ORDER);
455
456   u[0] = 0.5*(a0-b0);
457   u[1] = 0.5*(a1-b1);
458
459   for (k=0; k<GAUSS_ORDER; ++k){
460       /* transformation of quadrature nodes from [-1,1] to [a,b] */
461       sx0 = ((1-gauss_point[k])*c0+(1+gauss_point[k])*d0)*0.5;
462       sx1 = ((1-gauss_point[k])*c1+(1+gauss_point[k])*d1)*0.5;
463
464       v[0] = sx0 - 0.5*(a0+b0);
465       v[1] = sx1 - 0.5*(a1+b1);
466
467       /* inner product wht*func(sx) */
468       val += gauss_wht[k] * slp(0, u, v);
469   }
470
471   return -0.0625*val / M_PI; /* = - 1/(16*M_PI) * int(log |.|^2) */
472 }
```

4.1.7 `computeWijFullQuadrature`

Die Funktion gibt eine approximative Berechnung des Integrals (4.7) zurück, wobei beide Integrale mittels Gauß-Quadratur berechnet werden. Die Funktion berechnet also

$$-\frac{1}{16\pi}\sum_{k=0}^{p}\sum_{\ell=0}^{p}\omega_k\omega_\ell \log|\gamma_j(z_k)-\gamma_i(z_\ell)|^2,$$

wobei $p = \texttt{GAUSS_ORDER} - 1$.

Listing 7: computeWijFullQuadrature

```
625 double computeWijFullQuadrature(double a0, double a1, double b0,
626         double b1, double c0, double c1, double d0, double d1) {
627   /*
628    * INPUT:  elements Ti = [a,b], Tj = [c,d] with a,b,c,d \in \R^2
629    * OUTPUT: Galerkin integral
630    *         -1/(2pi) * 1/|Ti| * 1/|Tj| \int_{Tj} \int_{Ti} log|x-y| ds_y ds_x
631    *         where integration is performed by Gaussian quadrature
632    */
633   int k,l;
634   double u[2], v[2];
635   double val = 0;
636   double sx0 = 0;
637   double sx1 = 0;
638   double sy0 = 0;
639   double sy1 = 0;
640   const double* gauss_point;
641   const double* gauss_wht;
642
643   gauss_point = getGaussPoints(GAUSS_ORDER);
644   gauss_wht   = getGaussWeights(GAUSS_ORDER);
645
```

```
646     for (k=0; k<GAUSS_ORDER; ++k){
647         /* transformation of quadrature nodes from [-1,1] to [c,d] */
648         sx0 = ((1-gauss_point[k])*c0+(1+gauss_point[k])*d0)*0.5;
649         sx1 = ((1-gauss_point[k])*c1+(1+gauss_point[k])*d1)*0.5;
650
651         for (l=0; l<GAUSS_ORDER; ++l){
652             /* transformation of quadrature nodes from [-1,1] to [a,b] */
653             sy0 = ((1-gauss_point[l])*a0+(1+gauss_point[l])*b0)*0.5;
654             sy1 = ((1-gauss_point[l])*a1+(1+gauss_point[l])*b1)*0.5;
655
656             /* inner product wht*wht*func(sx,sy) */
657             val += gauss_wht[k] * gauss_wht[l] * log((sx0-sy0)*(sx0-sy0)+
658                    (sx1-sy1)*(sx1-sy1));
659         }
660     }
661
662     return -0.0625*val / M_PI; /* = - 1/(16*M_PI) * int(log |.|^2) */
663 }
```

4.1.8 computeVP1P1ij

Die Funktion erhält die Elemente $T_i = [\boldsymbol{a}, \boldsymbol{b}], T_j = [\boldsymbol{c}, \boldsymbol{d}]$ in Form ihrer Randpunkte $\boldsymbol{a} = [\mathtt{a0}, \mathtt{a1}]$, $\boldsymbol{b} = [\mathtt{b0}, \mathtt{b1}]$, $\boldsymbol{c} = [\mathtt{c0}, \mathtt{c1}]$ und $\boldsymbol{d} = [\mathtt{d0}, \mathtt{d1}]$. Zusätzlich wird die Zulässigkeitskonstante η übergeben. Darüber hinaus wird auch noch der Polynomgrad ℓ der Testfunktion χ_i^ℓ und der Grad k der Ansatzfunktion χ_j^k übergeben. Mit Hilfe des max-min-Kriteriums, welches in den Zeilen 513-530 realisiert ist, wird nun das Doppelintegral

$$-\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi} \int_{-1}^{1} \int_{-1}^{1} \log\left(|\gamma_i(s) - \gamma_j(t)|\right) t^\ell s^k dt\, ds \tag{4.8}$$

approximiert. Dabei wird sichergestellt, dass $\mathrm{diam}(T_i) \leq \mathrm{diam}(T_j)$. Andernfalls werden die beiden Elemente vertauscht. Gegebenenfalls müssen dann auch für die weitere Berechnung die Polynomgrade ℓ und k vertauscht werden.

Listing 8: computeVP1P1ij

```
474 double computeVP1P1ij(double a0, double a1, double b0, double b1, double c0,
475                       double c1, double d0, double d1, int l, int k,
476                       double eta) {
477
478     double u[2];
479     double v[2];
480     double w[2];
481     double hi = 0.0 , hj = 0.0;
482
483     double tmp = 0.0;
484     double val = 0.0;
485
486     int swapped = 0;
487
488     /* h_i = |T_i| */
489     /* h_j = |T_j| */
490     hi = ((b0-a0)*(b0-a0)+(b1-a1)*(b1-a1));
491     hj = ((c0-d0)*(c0-d0)+(c1-d1)*(c1-d1));
492
493      /* For stability reasons, we guarantee   hj >= hi   to ensure that *
494      * outer integration is over smaller domain. This is done by         *
495      * swapping Tj and Ti if necessary.                                  */
496
497     if (hj < hi) {
498       tmp = a0; a0 = c0; c0 = tmp;
499       tmp = a1; a1 = c1; c1 = tmp;
500       tmp = b0; b0 = d0; d0 = tmp;
501       tmp = b1; b1 = d1; d1 = tmp;
502       tmp = hi; hi = hj; hj = tmp;
503       swapped = 1;
```

```
504   }
505
506   if ( eta == 0) { /* compute all matrix entries analytically */
507     if(swapped)
508       val = computeVP1P1ijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,k,l);
509     else
510       val = computeVP1P1ijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,l,k);
511   }
512   else { /* compute admissible matrix entries semi-analytically */
513     if ( eta*distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1) > sqrt(hj) ) {
514         if(swapped)
515         val = computeVP1P1ijFullQuadrature(a0,a1, b0,b1, c0,c1, d0,d1,k,l);
516         else
517         val = computeVP1P1ijFullQuadrature(a0,a1, b0,b1, c0,c1, d0,d1,l,k);
518     }
519     else if ( eta*distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1) > sqrt(hi) ) {
520       if(swapped)
521         val = computeVP1P1ijSemiAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,k,l);
522       else
523         val = computeVP1P1ijSemiAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,l,k);
524     }
525     else {
526       if(swapped)
527         val = computeVP1P1ijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,k,l);
528       else
529         val = computeVP1P1ijAnalytic(a0,a1, b0,b1, c0,c1, d0,d1,l,k);
530     }
531   }
532
533
534   /* return -1/(2pi) |T_i|/2*|T_j|/2 int_-1^1 int_-1^1 ... */
535   return -0.125 / M_PI *sqrt(hi*hj)* val;
536 }
```

4.1.9 computeVP1P1ijAnalytic

Die Funktion berechnet nun das Integral

$$\int_{-1}^{1}\int_{-1}^{1} \log\left(|\gamma_i(s) - \gamma_j(t)|\right) t^\ell s^k dt\ ds. \tag{4.9}$$

Der wesentliche Teil der Berechnung wird dabei allerdings in der Funktion `doubleSlp` realisiert.

Listing 9: computeVP1P1ijAnalytic

```
587 double computeVP1P1ijAnalytic(double a0, double a1, double b0, double b1,
588                                  double c0, double c1, double d0, double d1,
589                              int l, int k) {
590
591   double val = 0.0;
592   double u[2];
593   double v[2];
594   double w[2];
595
596     /* u = (z_i,2 -z_i,1)/2  */
597   u[0] = 0.5*(b0-a0);
598   u[1] = 0.5*(b1-a1);
599   /* v = -(z_j,2 - z_j,1)/2 */
600   v[0] = 0.5*(c0-d0);
601   v[1] = 0.5*(c1-d1);
602   /* w = (z_i,2+z_i,1)/2 - (z_j,2+z_j,1)/2 */
603   w[0] = 0.5*(a0+b0-c0-d0);
604   w[1] = 0.5*(a1+b1-c1-d1);
605   /*
606   if(l==1) {
607       if(k==1)
608         val = doubleSlp(1,1,u,v,w);
609       else if(k==0)
```

```
610          val = doubleSlp(1,0,u,v,w);
611      }
612      else if (l==0) {
613        if(k==1) {
614          val = doubleSlp(0,1,u,v,w);
615        }
616         else if(k==0)
617           val = doubleSlp(0,0,u,v,w);
618      }*/
619      val = doubleSlp(1,k,u,v,w);
620
621      return val;
622    }
```

4.1.10 computeVP1P1ijSemiAnalytic

Die Funktion berechnet nun das Integral (4.9) semianalytisch, das heißt es wird für die übergebenen Werte $m, n \in \{0, 1\}$ das Integral

$$\frac{1}{2}\sum_{k=0}^{p}\omega_k\int_{-1}^{1}\log\left(|\gamma_i(z_k)-\gamma_j(t)|\right)^2 t^m z_k^n dt$$

berechnet, wobei $p = \texttt{GAUSS_ORDER} - 1$.

Listing 10: computeVP1P1ijSemiAnalytic

```
538  double computeVP1P1ijSemiAnalytic(double a0, double a1, double b0, double b1,
539                                    double c0, double c1, double d0, double d1,
540                                    int m, int n) {
541    int k,i;
542    double u[2], v[2];
543    double val = 0;
544    double sx0 = 0;
545    double sx1 = 0;
546    const double* gauss_point;
547    const double* gauss_wht;
548
549    gauss_point = getGaussPoints(GAUSS_ORDER);
550    gauss_wht   = getGaussWeights(GAUSS_ORDER);
551    /*
552    u[0] = 0.5*(a0-b0);
553    u[1] = 0.5*(a1-b1);*/
554
555    u[0] = 0.5*(c0-d0);
556    u[1] = 0.5*(c1-d1);
557
558    for (k=0; k<GAUSS_ORDER; ++k){
559        /* transformation of quadrature nodes from [-1,1] to [a,b] */
560        sx0 = ((1-gauss_point[k])*a0+(1+gauss_point[k])*b0)*0.5;
561        sx1 = ((1-gauss_point[k])*a1+(1+gauss_point[k])*b1)*0.5;
562
563        v[0] = sx0 - 0.5*(c0+d0);
564        v[1] = sx1 - 0.5*(c1+d1);
565
566        /* inner product wht*func(sx) */
567        if(m==0) {
568          if(n==1) {
569            val += gauss_wht[k] * slp(1,u,v);
570          }
571          else if(n==0) {
572            val += gauss_wht[k] * slp(0,u,v);
573          }
574        }
575        else if(m==1) {
576          if(n==1)
577            val += gauss_wht[k] * gauss_point[k] * slp(1,u,v);
578          else if(n==0)
579            val += gauss_wht[k] * gauss_point[k] * slp(0,u,v);
```

```
580         }
581
582     }
583
584     return 0.5*val; /* = ... */
585 }
```

4.1.11 computeVP1P1ijFullQuadrature

Die Funktion berechnet nun das Integral (4.9) indem beide Integrale durch Quadratur ersetzt werden. Für die übergebenen Werte $m, n \in \{0,1\}$ wird der Wert

$$\frac{1}{2}\sum_{k=0}^{p}\sum_{\ell=0}^{p}\omega_k\omega_\ell \log\left(|\gamma_i(z_k)-\gamma_j(z_\ell)|\right)^2 z_\ell^m z_k^n$$

berechnet, wobei $p = \texttt{GAUSS_ORDER} - 1$.

Listing 11: computeVP1P1ijFullQuadrature

```
666 double computeVP1P1ijFullQuadrature(double a0, double a1, double b0,
667         double b1, double c0, double c1, double d0, double d1, int m,
668         int n) {
669   int k,l;
670   double val = 0;
671   double sx0 = 0;
672   double sx1 = 0;
673   double sy0 = 0;
674   double sy1 = 0;
675   const double* gauss_point;
676   const double* gauss_wht;
677
678   gauss_point = getGaussPoints(GAUSS_ORDER);
679   gauss_wht   = getGaussWeights(GAUSS_ORDER);
680
681   for (k=0; k<GAUSS_ORDER; ++k){
682       /* transformation of quadrature nodes from [-1,1] to [a,b] */
683       sx0 = ((1-gauss_point[k])*a0+(1+gauss_point[k])*b0)*0.5;
684       sx1 = ((1-gauss_point[k])*a1+(1+gauss_point[k])*b1)*0.5;
685
686       for (l=0; l<GAUSS_ORDER; ++l){
687           /* transformation of quadrature nodes from [-1,1] to [c,d] */
688           sy0 = ((1-gauss_point[l])*c0+(1+gauss_point[l])*d0)*0.5;
689           sy1 = ((1-gauss_point[l])*c1+(1+gauss_point[l])*d1)*0.5;
690
691           /* inner product wht*wht*func(sx,sy) */
692           if(m==0) {
693             if(n==1) {
694               val += gauss_wht[k] * gauss_wht[l] * gauss_point[l] *
695                      log((sx0-sy0)*(sx0-sy0)+(sx1-sy1)*(sx1-sy1));
696             }
697             else if(n==0) {
698               val += gauss_wht[k] * gauss_wht[l] * log((sx0-sy0)*
699                      (sx0-sy0)+(sx1-sy1)*(sx1-sy1));
700             }
701           }
702           else if(m==1) {
703             if(n==1)
704               val += gauss_wht[k] * gauss_point[k] * gauss_wht[l] *
705                      gauss_point[l] * log((sx0-sy0)*(sx0-sy0)+(sx1-sy1)*
706                      (sx1-sy1));
707             else if(n==0)
708               val += gauss_wht[k] * gauss_point[k] * gauss_wht[l] *
709                      log((sx0-sy0)*(sx0-sy0)+(sx1-sy1)*(sx1-sy1));
710           }
711       }
712   }
713
714   return 0.5*val; /* = ... */
715 }
```

4.1.12 slp

Die Funktion dient zur Berechnung von

$$\int_{-1}^{1} s^k \log |s\boldsymbol{u} + \boldsymbol{v}|^2 \, ds \quad \text{für } k \in \mathbb{N}_0 \text{ und } \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^2. \tag{4.10}$$

Die eigentliche Berechnung passiert allerdings in `slpIterative`, welche einen Array zurückgibt. `slp` extrahiert eigentlich nur den letzten Eintrag aus diesem Array.

Listing 12: slp

```
20 double slp(int k, double u[2], double v[2]) {
21   double ret = 0.;
22   double* tmp = slpIterative(k, u, v);
23   ret = tmp[k];
24   free(tmp);
25   return ret;
26 }
```

4.1.13 slpIterative

Die Funktion berechnet nun tatsächlich das Integral (4.10). Da diese Berechnung rekursiv stattfindet, wird gleich ein ganzer Array der Dimension $k+1$ erstellt, dessen Einträge durch

$$\texttt{val[i]} = \int_{-1}^{1} s^i \log |s\boldsymbol{u} + \boldsymbol{v}|^2 \, ds \quad \text{für } i = 0, \ldots, k. \tag{4.11}$$

für $k \in \mathbb{N}_0$ und $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^2$ gegeben sind. Für Details zur Berechnung möchten wir auf [10, Lemma 2.2] verweisen.

Listing 13: slpIterative

```
28 double* slpIterative(int k, double u[2], double v[2]) {
29   /* INPUT: Vectors \u, \v \in \R^2, an integer k.
30    * OUTPUT: An array val of length k+1 that is given by
31    *            val[i] = int_{-1}^1 s^i log |su+v|^2 ds
32    */
33   int i = 0;
34   double a = u[0]*u[0] + u[1]*u[1];         /* a = <u,u> */
35   double b = 2 * ( u[0]*v[0] + u[1]*v[1] ); /* b = 2 <u,v> */
36   double c = v[0]*v[0] + v[1]*v[1];         /* c = <v,v> */
37   double tmp = 0., D = 0.;
38   double* val = malloc(sizeof(double)*(k+1));
39 
40   /* Ensure that discriminant is either positive or zero */
41   tmp = 4*a*c - b*b;
42   assert(fabs(u[0]) > EPS || fabs(u[1]) > EPS
43          || fabs(v[0]) > EPS || fabs(v[1]) > EPS);
44   assert(tmp >= -fabs(EPS*4*a*c)); /* By theory there holds tmp >= 0. */
45 
46   if (tmp > EPS*4*a*c)
47     D = sqrt(tmp);
48   else
49     D = 0.;
50 
51   /* The case k=0 */
52   if (fabs(u[0]) < EPS && fabs(u[1]) < EPS) {
53       val[0] = 2*log(c);
54   }
55   else if (D == 0.) {
56     tmp = b + 2*a;
57     if (fabs(tmp) > EPS*a)
58       val[0] = tmp * log( 0.25*tmp*tmp /a );
```

```
59      else
60        val[0] = 0;
61      tmp = b - 2*a;
62      if (fabs(tmp) > EPS*a)
63        val[0] -= tmp * log( 0.25*tmp*tmp /a );
64      val[0] = 0.5*val[0] /a - 4;
65    }
66    else { /* case D > 0 */
67      tmp = c - a;
68      if (fabs(tmp) < EPS*c)
69        val[0] = 0.5*M_PI;
70      else if (a < c)
71        val[0] = atan( D /tmp );
72      else
73        val[0] = atan( D /tmp ) + M_PI;
74
75      val[0] = ( 0.5*( (b+2*a) * log(a+b+c) - (b-2*a) * log(a-b+c) ) + D*val[0])
76                                                                        / a - 4;
77    }
78    if (k == 0)
79      return val;
80
81    /* The case k=1 */
82    if (k>=1) {
83      if (fabs(u[0]) < EPS && fabs(u[1]) < EPS) {
84        val[1] = 0.;
85      }
86      else {
87        /* val holds \int_{-1}^{+1} \log |a*s^2+b*s+c|^2 ds. */
88        val[1] = -b*(2+val[0]);
89
90        tmp = a+b+c;
91        if (fabs(tmp) > EPS*a)
92          val[1] += tmp * log(tmp);
93
94        tmp = a-b+c;
95        if (fabs(tmp) > EPS*a)
96          val[1] -= tmp * log(tmp);
97
98        val[1] /= (2*a);
99      }
100   }
101   if (k == 1)
102     return val;
103
104   /* The case k>=2 */
105   for (i=2; i <= k; ++i) {
106     if (fabs(u[0]) < EPS && fabs(u[1]) < EPS) {
107       if (i%2 == 0)
108         val[i] = 2*log(c)/(double)(i+1);
109       else
110         val[i] = 0.;
111     }
112     else {
113       tmp = a+b+c;
114       if (fabs(tmp) > a*EPS)
115         val[i] = tmp*log(tmp);
116       else
117         val[i] = 0.;
118
119       tmp = a-b+c;
120       if (i % 2 == 0) {
121         if (fabs(tmp) > a*EPS)
122           val[i] += tmp*log(tmp);
123         val[i] -= 4*a/(i+1);
124       }
125       else {
126         if (fabs(tmp) > a*EPS)
127           val[i] -= tmp*log(tmp);
128         val[i] -= 2*b/i;
129       }
130
131       val[i] -= i*b*val[i-1]+(i-1)*c*val[i-2];
132       val[i] /= ((i+1)*a);
```

```
133     }
134   }
135 
136   return val;
137 }
```

4.1.14 `doubleSlp`

Für $k, \ell \in \mathbb{N}_0$ und $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^2$ berechnet die Funktion

$$\int_{-1}^{1} \int_{-1}^{1} s^k t^\ell \log |s\boldsymbol{u} + t\boldsymbol{v} + \boldsymbol{w}| \, dt \, ds.$$

Die Berechnung lässt sich rekursiv und mit Hilfe der Funktion `slpIterative` realisieren. Für Details sei auch hier auf [10, Lemma 3.1] verwiesen.

Listing 14: doubleSlp

```
139 double doubleSlp(int k, int l, double u[], double v[], double w[]) {
140   int i = 1;
141   double output = 0.;
142   double wpv[2], wmv[2], wpu[2], wmu[2];
143   double* memTableUWpv = NULL;
144   double* memTableUWmv = NULL;
145   double* memTableVWpu = NULL;
146   double* memTableVWmu = NULL;
147   double normUSq = u[0]*u[0]+u[1]*u[1];
148   double normVSq = v[0]*v[0]+v[1]*v[1];
149   double normWSq = w[0]*w[0]+w[1]*w[1];
150   double detUV = u[0]*v[1] - u[1]*v[0];
151 
152   if (sqrt(normUSq) < EPS && sqrt(normVSq) < EPS) {
153     mexWarnMsgTxt("||u||␣&&␣||v||␣<␣EPS");
154     if (k%2 == 0) {
155       mexWarnMsgTxt("k␣mod␣2␣==␣0");
156       if (l%2 == 0) {
157           mexWarnMsgTxt("l␣mod␣2␣==␣0");
158           if (normWSq < EPS*EPS) {
159               mexErrMsgTxt("||w||<EPS");
160               return 0;
161           }
162       }
163       else
164           return (double)2./(double)((k+1)*(l+1))*log(w[0]*w[0]+w[1]*w[1]);
165 
166     }
167     return 0.;
168   }
169   else if (normUSq < EPS*EPS) {
170     mexWarnMsgTxt("||u||␣<␣EPS");
171     if (k%2 == 0)
172       return (double) 1./(k+1) * slp(l, v, w);
173     return 0.;
174   }
175   else if (normVSq < EPS*EPS) {
176     mexWarnMsgTxt("||w||<EPS");
177     if (l%2 == 0)
178       return (double) 1./(l+1) * slp(k, u, w);
179     return 0.;
180   }
181 
182   wpv[0] = w[0] + v[0]; wpv[1] = w[1] + v[1];
183   wmv[0] = w[0] - v[0]; wmv[1] = w[1] - v[1];
184   wpu[0] = w[0] + u[0]; wpu[1] = w[1] + u[1];
185   wmu[0] = w[0] - u[0]; wmu[1] = w[1] - u[1];
186 
187   if (fabs(detUV) < EPS*sqrt(normUSq*normVSq)) { /* u,v parallel */
188     double mu = 0.;
189 
```

```
190       memTableUWpv = slpIterative(k, u, wpv);
191       memTableUWmv = slpIterative(k, u, wmv);
192       memTableVWpu = slpIterative(k+l+1, v, wpu);
193       memTableVWmu = slpIterative(k+l+1, v, wmu);
194
195       if (fabs(u[0]) < fabs(u[1]))
196         mu = v[1] / u[1];
197       else
198         mu = v[0] / u[0];
199
200       output = memTableUWpv[0] - mu*memTableVWpu[k+l+1] + mu*memTableVWmu[k+l+1];
201       if ((k+l) % 2 == 0)
202         output += memTableUWmv[0];
203       else
204         output -= memTableUWmv[0];
205       output /= (2*(k+l+1));
206
207       for (i = 1; i <= k; ++i) {
208         output *= 2*i*mu;
209         output += memTableUWpv[i] - mu*memTableVWpu[l+k+1-i];
210
211         if ((k+l-i) % 2 == 0)
212           output += memTableUWmv[i];
213         else
214           output -= memTableUWmv[i];
215
216         if (i % 2 == 0)
217           output += mu*memTableVWmu[l+k+1-i];
218         else
219           output -= mu*memTableVWmu[l+k+1-i];
220
221         output /= (2*(l+k+1-i));
222       }
223     }
224     else {
225       int j = 1;
226       double mu1 = 0., mu2 = 0.;
227       double* tmp = malloc(sizeof(double) * (l+1));
228
229       if (fabs(w[0]+v[0]-u[0]) < fabs(w[0])*EPS
230             && fabs(w[1]+v[1]-u[1]) < fabs(w[1])*EPS) {
231         memTableVWmu = malloc(sizeof(double) * (l+1));
232         for (i = 0; i <= l; ++i)
233           memTableVWmu[i] = 0.;
234
235         memTableUWpv = malloc(sizeof(double) * (k+1));
236         for (i = 0; i <= k; ++i)
237           memTableUWpv[i] = 0.;
238       }
239       else {
240         memTableVWmu = slpIterative(l, v, wmu);
241         memTableUWpv = slpIterative(k, u, wpv);
242       }
243
244       if (fabs(w[0]+u[0]-v[0]) < fabs(w[0])*EPS
245             && fabs(w[1]+u[1]-v[1]) < fabs(w[1])*EPS) {
246         memTableVWpu = malloc(sizeof(double) * (l+1));
247         for (i = 0; i <= l; ++i)
248           memTableVWpu[i] = 0.;
249
250         memTableUWmv = malloc(sizeof(double) * (k+1));
251         for (i = 0; i <= k; ++i)
252           memTableUWmv[i] = 0.;
253       }
254       else {
255         memTableVWpu = slpIterative(l, v, wpu);
256         memTableUWmv = slpIterative(k, u, wmv);
257       }
258
259       mu1 = ( v[1]*w[0] - v[0]*w[1]) / detUV;
260       mu2 = (-u[1]*w[0] + u[0]*w[1]) / detUV;
261
262       tmp[0] = -2 + ((mu1+1)*memTableVWpu[0] - (mu1-1)*memTableVWmu[0]
263                 + (mu2+1)*memTableUWpv[0] - (mu2-1)*memTableUWmv[0]) * 0.25;
```

```
264
265      for (i = 1; i <= l; ++i) {
266        tmp[i] = 0.5*((mu1+1)*memTableVWpu[i] - (mu1-1)*memTableVWmu[i]
267              + (mu2+1)*memTableUWpv[0]) - i*mu2*tmp[i-1];
268        if (i%2 == 0) {
269          tmp[i] -= (double)4/(double)(i+1);
270          tmp[i] -= 0.5 * (mu2-1)*memTableUWmv[0];
271        }
272        else
273          tmp[i] += 0.5 * (mu2-1)*memTableUWmv[0];
274
275        tmp[i] /= (i+2.);
276      }
277
278      for (i = 1; i <= k; ++i) {
279        tmp[0] = 0.5*((mu1+1)*memTableVWpu[0] + (mu2+1)*memTableUWpv[i]
280                    - (mu2-1)*memTableUWmv[i]) - i*mu1*tmp[0];
281        if (i%2 == 0) {
282          tmp[0] -= (double)4/(double)(i+1);
283          tmp[0] -= 0.5*(mu1-1)*memTableVWmu[0];
284        }
285        else {
286          tmp[0] += 0.5*(mu1-1)*memTableVWmu[0];
287        }
288
289        tmp[0] /= (i+2.);
290
291        for (j = 1; j <= l; ++j) {
292          tmp[j] *= -i*mu1;
293          tmp[j] -= j*mu2*tmp[j-1];
294          tmp[j] += 0.5*( (mu1+1)*memTableVWpu[j] + (mu2+1)*memTableUWpv[i] );
295          if (i%2 == 0) {
296            if (j%2 == 0) {
297              tmp[j] -= (double)4/(double)((i+1)*(j+1));
298            }
299            tmp[j] -= 0.5 * (mu1-1) * memTableVWmu[j];
300          }
301          else {
302            tmp[j] += 0.5 * (mu1-1) * memTableVWmu[j];
303          }
304
305          if (j%2 == 0) {
306            tmp[j] -= 0.5 * (mu2-1) * memTableUWmv[i];
307          }
308          else {
309            tmp[j] += 0.5 * (mu2-1) * memTableUWmv[i];
310          }
311
312          tmp[j] /= (i+j+2.);
313        }
314      }
315
316      output = tmp[l];
317      free(tmp);
318    }
319
320    free(memTableUWpv);
321    free(memTableUWmv);
322    free(memTableVWpu);
323    free(memTableVWmu);
324
325    return output;
326  }
```

4.2 Implementierung des K-Operators

Auch bei der Implementierung des K-Operators ist die C-Schnittstelle zu MATLAB durch die `mexFunction` gegeben, welche die Eingabeparameter einliest und den Speicher für die Matrix $\mathbf{K}$ allokiert. Die Funktion `computeKP1S2` organisiert dann die Berechnung der

Einträge:

$$\mathbf{K}_{ij}^{(\ell m)} = -\frac{1}{2\pi}\int_{T_i}\int_{\mathrm{supp}(\zeta_j^m)} \frac{(\boldsymbol{y}-\boldsymbol{x})\cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2}\zeta_j^m(\boldsymbol{y})\chi_i^\ell(\boldsymbol{x})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \tag{4.12}$$

für $\ell \in \{0,1\}$ und $m \in \{1,2\}$.

Die Berechnung der Einträge für $m = 1$ läuft etwas anders als beim Einfachschichtpotential-Operator ab, da in diesem Fall $\mathrm{supp}(\zeta_j^m)$ mehr als nur ein Element umfasst. Wir möchten an dieser Stelle nochmals hervorheben, dass j in diesem Fall die Menge der Knoten durchläuft, also $j \in \{1,\dots,\widetilde{N}\}$. Sei für einen festen Eintrag $\mathbf{K}_{ij}^{(\ell 1)}$ der Matrix $\mathbf{K}$ die zugehörige Basisfunktion $\zeta_j^1 \in \mathcal{S}^1(\mathcal{T}_h)$, die zu einem bestimmten Knoten $\boldsymbol{z}_j$ gehört. Mit Hilfe der Darstellung von ζ_j^1 aus (3.9) und $\mathbf{K}_{ij}^{(\ell 1)}$ aus (3.23) gilt nun

$$\begin{aligned}
\mathbf{K}_{ij}^{(\ell 1)} &= -\frac{1}{2\pi}\int_{T_i}\int_{T_{j_1}} \frac{(\boldsymbol{y}-\boldsymbol{x})\cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2}\zeta_j^1(\boldsymbol{y})\chi_i^\ell(\boldsymbol{x})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \\
&\quad - \frac{1}{2\pi}\int_{T_i}\int_{T_{j_2}} \frac{(\boldsymbol{y}-\boldsymbol{x})\cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2}\zeta_j^1(\boldsymbol{y})\chi_i^\ell(\boldsymbol{x})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \\
&= -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_{j_1})}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_{j_1}(t)-\gamma_i(s))\cdot n(\gamma_{j_1}(s))}{|\gamma_i(s)-\gamma_{j_1}(t)|^2}\left(\frac{1+t}{2}\right)s^\ell dt\,ds \\
&\quad - \frac{\mathrm{diam}(T_i)\mathrm{diam}(T_{j_2})}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_{j_2}(t)-\gamma_i(s))\cdot n(\gamma_{j_2}(s))}{|\gamma_i(s)-\gamma_{j_2}(t)|^2}\left(\frac{1-t}{2}\right)s^\ell dt\,ds.
\end{aligned}$$

Bei der Implementierung in der Funktion `computeKP1S2` wird elementweise vorgegangen. Es wird also über alle Elemente iteriert. Da jedes Element genau zwei Randknoten hat, wird im entsprechenden Matrixeintrag $\mathbf{K}_{ij}^{\ell 1}$ für die Randknoten das dem Element entsprechende Doppelintegral addiert. Im Wesentlichen treten dabei für ein Element $T_{\widetilde{j}}$ zwei Integrale auf, die es zu berechnen gilt:

$$\begin{aligned}
I_{0,i,\widetilde{j}}^\ell &:= -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_{\widetilde{j}})}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_{\widetilde{j}}(t)-\gamma_i(s))\cdot n(\gamma_{\widetilde{j}}(t))}{|\gamma_i(s)-\gamma_{\widetilde{j}}(t)|^2}\frac{1}{2}s^\ell dt\,ds \\
I_{1,i,\widetilde{j}}^\ell &:= -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_{\widetilde{j}})}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_{\widetilde{j}}(t)-\gamma_i(s))\cdot n(\gamma_{\widetilde{j}}(t))}{|\gamma_i(s)-\gamma_{\widetilde{j}}(t)|^2}\frac{t}{2}s^\ell dt\,ds.
\end{aligned} \tag{4.13}$$

Damit erhalten wir nun

$$\mathbf{K}_{ij}^{(\ell 1)} = I_{0,i,j_1}^\ell + I_{1,i,j_1}^\ell + I_{0,i,j_2}^\ell - I_{1,i,j_2}^\ell. \tag{4.14}$$

Für den Fall $m = 2$ läuft die Berechnung der Einträge $\mathbf{K}_{ij}^{(\ell 2)}$ wieder so ähnlich wie beim V-Operator ab. Für $j \in \{1,\dots,N\}$ mit $\zeta_j^2(\gamma_j(t)) = 1/4(1-t^2)$ gilt also

$$\begin{aligned}
\mathbf{K}_{ij}^{(\ell 2)} &= -\frac{1}{2\pi}\int_{T_i}\int_{T_j} \frac{(\boldsymbol{y}-\boldsymbol{x})\cdot n(\boldsymbol{y})}{|\boldsymbol{x}-\boldsymbol{y}|^2}\zeta_j^2(\boldsymbol{y})\chi_i^\ell(\boldsymbol{x})ds_{\boldsymbol{y}}ds_{\boldsymbol{x}} \\
&= -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{32\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(s))\cdot n(\gamma_j(t))}{|\gamma_i(s)-\gamma_j(t)|^2}(1-t^2)s^\ell dt\,ds.
\end{aligned} \tag{4.15}$$

für $\ell \in \{0,1\}$.

Für die Einträge der Submatrix $\mathbf{K}^{(01)}$ (vgl.(3.17)) werden nun in der Funktion `computeKij` die Doppelintegrale $I^0_{0,i,\widetilde{j}}$ und $I^0_{1,i,\widetilde{j}}$ unter Anwendung des max-min-Kriteriums für alle Elemente $T_i, T_{\widetilde{j}}$ berechnet. Für den Fall, dass die Elemente $T_i, T_{\widetilde{j}}$ η-unzulässig sind oder die Zulässigkeitskonstante η gleich 0 ist, so werden die Doppelintegrale $I^0_{0,i,\widetilde{j}}$ und $I^0_{1,i,\widetilde{j}}$ analytisch berechnet. Da das Integral über das längere Randstück zuerst berechnet werden soll und anschließend das Integral über das kürzere Randstück, wird im Fall $\operatorname{diam}(T_i) \leq \operatorname{diam}(T_{\widetilde{j}})$ die Funktion `computeKijAnalytic` aufgerufen, und ansonsten die Funktion `computeKijAnalyticSwapped`. Falls die Elemente $T_i, T_{\widetilde{j}}$ η-min-zulässig sind, wird für $\operatorname{diam}(T_i) \leq \operatorname{diam}(T_{\widetilde{j}})$ die Funktion `computeKijSemianalytic` aufgerufen und für $\operatorname{diam}(T_{\widetilde{j}}) < \operatorname{diam}(T_i)$ die Funktion `computeKijSemianalyticSwapped`. Wenn die Elemente $T_i, T_{\widetilde{j}}$ η-max-zulässig sind, berechnet die Funktion `computeKijFullQuadrature` die Doppelintegrale. Dabei ist eine Unterscheidung, welches der beiden Elemente $T_i, T_{\widetilde{j}}$ den kleineren Durchmesser hat, nicht notwendig. Für ein Doppelintegral werden beide Integrale durch Quadratur ersetzt. Dabei werden nur Terme aufsummiert und daher spielt die Reihenfolge keine Rolle.
Für die analytische und semianalytische Berechnung der Doppelintegrale wird dann die exakte Berechnung von Integralen der Form

$$\int_{-1}^{1} \frac{t^k}{|t\boldsymbol{p} + \boldsymbol{q}|^2} dt \quad \text{für } k \in \mathbb{N}_0 \text{ und } \boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^2$$

benötigt. Dies wird in der Funktion `dlp` realisiert.

Für die anderen Submatrizen $\mathbf{K}^{(02)}$, $\mathbf{K}^{(11)}$ und $\mathbf{K}^{(12)}$ werden die Einträge mit der Funktion `computeKP1S2ij` berechnet. Bei der Submatrix $\mathbf{K}^{(11)}$ werden nun für alle Elemente $T_i, T_{\widetilde{j}}$ die Doppelintegrale $I^1_{0,i,\widetilde{j}}$ und $I^1_{1,i,\widetilde{j}}$ unter Anwendung des max-min-Kriteriums approximiert. Für die anderen zwei Submatrizen $\mathbf{K}^{(02)}$ und $\mathbf{K}^{(12)}$ wird nun für jeden Eintrag eine Approximation für das Doppelintegral (4.15) mit Hilfe des max-min-Kriteriums berechnet. In der Funktion `computeKP1S2ij` wird überprüft, ob die zwei Elemente T_i, T_j über die integriert werden soll, eine Zulässigkeitsbedingung erfüllen. Falls diese η-unzulässig sind oder die Zulässigkeitskonstante η gleich 0 ist wird das geforderte Doppelintegral analytisch mit der Funktion `computeKP1S2ijAnalytic` berechnet. Für den Fall, dass die Elemente T_i, T_j η-min-zulässig sind, wird die Funktion `computeKP1S2ijSemianalytic` aufgerufen. Wenn T_i und T_j sogar η-max-zulässig sind, dann wird die Berechnung in der Funktion `computeKP1S2ijFullQuadrature` mittels Quadratur durchgeführt.
Für die analytische Berechnung der Doppelintegrale werden in weiterer Folge die Funktionen `doubleDlp` beziehungsweise `doubleDlpSwapped` benötigt, welche Integrale der Form

$$\int_{-1}^{1}\int_{-1}^{1} s^k t^\ell \frac{(\boldsymbol{w} + s\boldsymbol{u}) \cdot n_{\boldsymbol{v}}}{|\boldsymbol{w} + s\boldsymbol{u} + t\boldsymbol{v}|^2} \, dt \, ds \quad \text{für } k, \ell \in \mathbb{N}_0 \text{ und } \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^2$$

beziehungsweise

$$\int_{-1}^{1}\int_{-1}^{1} s^k t^\ell \frac{(\boldsymbol{w} + t\boldsymbol{v}) \cdot n_{\boldsymbol{u}}}{|\boldsymbol{w} + s\boldsymbol{u} + t\boldsymbol{v}|^2} \, dt \, ds \quad \text{für } k, \ell \in \mathbb{N}_0 \text{ und } \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^2$$

berechnen.

4.2.1 `mexFunction`

Als Eingabeparameter erhält die Funktion die Arrays `coordinates` und `elements`, welche die Triangulierung $\mathcal{T}_h$ beschreiben. Wie bei der `mexFunction` der Implementierung des V-

Operators stehen in jeder Zeile der $\widetilde{N} \times 2$-Matrix `coordinates` die Koordinaten eines Randknotens aus $\mathbb{R}^2$. Die $N \times 2$-Matrix `elements` beschreibt die Randelemente, wobei in jeder Zeile die Indizes für die Zeile der zugehörigen Koordinaten aus `coordinates` stehen. Optional kann auch noch die Zulässigkeitskonstante η übergeben werden. Falls dies nicht geschieht, wird dafür der vordefinierte Wert `DEFAULT_ETA` verwendet.

Listing 15: mexFunction

```
25 void mexFunction(int nlhs, mxArray** plhs, int nrhs, const mxArray** prhs){
26   const char* functionName = mexFunctionName();
27   char errorMessage[150];
28   int nC=0, nE=0;
29   double eta = DEFAULT_ETA;
30   double* K=NULL;
31   const double* elements=NULL;
32   const double* coordinates=NULL;
33 
34   if(nlhs!=1){
35     sprintf(errorMessage,"Invalid␣number␣of␣output␣arguments.␣Use␣either\n"
36             "␣␣K␣=␣%s(coordinates,elements)␣or\n"
37             "␣␣K␣=␣%s(coordinates,elements,eta).\n",
38             functionName,functionName);
39     mexErrMsgTxt(errorMessage);
40   }
41 
42   /* read input data */
43   switch(nrhs){
44     case 3:
45       eta = (double) mxGetScalar(prhs[2]);
46     case 2:
47       coordinates = (const double*) mxGetPr(prhs[0]);
48       elements = (const double*) mxGetPr(prhs[1]);
49       nC = mxGetM(prhs[0]);
50       nE = mxGetM(prhs[1]);
51       break;
52     default:
53       sprintf(errorMessage,"Invalid␣number␣of␣input␣arguments.␣Use␣either\n"
54             "␣␣K␣=␣%s(coordinates,elements)␣or\n"
55             "␣␣K␣=␣%s(coordinates,elements,eta).\n",
56             functionName,functionName);
57       mexErrMsgTxt(errorMessage);
58       break;
59   }
60 
61   /* allocate output data */
62   plhs[0] = mxCreateDoubleMatrix(2*nE,nC+nE,mxREAL);
63   K = mxGetPr(plhs[0]);
64 
65   computeKP1S2(K,coordinates,elements,nC,nE,eta);
66 }
```

4.2.2 `computeKP1S2`

Als Eingabeparameter werden die Arrays `coordinates` und `elements` und deren Dimensionen übergeben, also die Elementanzahl N und die Anzahl der Randknoten $\widetilde{N}$. Zusätzlich werden noch ein Pointer, der auf den für die Matrix $\mathbf{K}$ allokierten Speicher zeigt, und die Zulässigkeitskonstante η übergeben. Die Funktion durchläuft nun alle Kombinationen von Elementen $T_i \times T_j$ und berechnet dann entsprechende Doppelintegrale, die für die Einträge von $\mathbf{K}$ notwendig sind.
Für die Submatrizen $\mathbf{K}^{(02)}$ und $\mathbf{K}^{(12)}$ wird dabei die Funktion `computeKP1S2ij` aufgerufen, um die entsprechenden Matrixeinträge $\mathbf{K}^{(\ell 2)}_{ij}$ (siehe (4.15)) mit $\ell = 0, 1$ zu berechnen.
Für die Submatrizen $\mathbf{K}^{(01)}, \mathbf{K}^{(11)} \in \mathbb{R}^{N \times \widetilde{N}}$ läuft die Berechnung etwas anders ab, da die Spalten nicht nach Elementen, sondern nach Randknoten indiziert sind. Für die Kombination $T_i \times T_j$ mit $T_j = [\boldsymbol{c}, \boldsymbol{d}]$ wird für die Einträge der Submatrix $\mathbf{K}^{(01)}$ die Funktion

computeKij aufgerufen. Diese Funktion berechnet dann die Doppelintegrale $I^0_{0,i,j}$ und $I^0_{1,i,j}$ aus (4.13). Im Eintrag der zum Randknoten $\boldsymbol{c}$ gehört, wird dann $I^0_{0,i,j} - I^0_{1,i,j}$ addiert, wobei im Eintrag der zu $\boldsymbol{d}$ gehört $I^0_{0,i,j} + I^0_{1,i,j}$ addiert wird. Für die Submatrix $\mathbf{K}^{(11)}$ werden dann $I^1_{0,i,j}$ und $I^1_{1,i,j}$ aus (4.13) mit Hilfe der Funktion computeKP1S2ij berechnet. Im Eintrag der zum Randknoten $\boldsymbol{c}$ gehört, wird dann $I^1_{0,i,j} - I^1_{1,i,j}$ addiert, wobei im Eintrag der zu $\boldsymbol{d}$ gehört $I^1_{0,i,j} + I^1_{1,i,j}$ addiert wird. Da alle Elemente durchlaufen werden, wird auf diese Art und Weise sichergestellt, dass für alle Knoten $\boldsymbol{z}_j$ mit $j \in \{1, \dots, \widetilde{N}\}$ die Einträge $\mathbf{K}^{\ell 1}_{ij}$ wie in (4.14) berechnet werden.

Listing 16: computeKP1S2

```
70  void computeKP1S2(double* K, const double* coordinates,
71                    const double* elements, int nC, int nE, double eta) {
72
73    int i=0, j=0;
74    int aidx=0, bidx=0, cidx=0, didx=0;
75    double a0=0.0, a1=0.0, b0=0.0, b1=0.0, c0=0.0, c1=0.0, d0=0.0, d1=0.0;
76    double linetest1=0.0, linetest2=0.0;
77    double I0=0.0, I1=0.0;
78    double J0=0.0, J1=0.0;
79
80    for (i=0;i<nE;++i) {
81      aidx = (int) elements[i]-1;                 /* 1st node of Ti = [A,B] */
82      a0 = coordinates[aidx];
83      a1 = coordinates[aidx+nC];
84
85      bidx = (int) elements[i+nE]-1;              /* 2nd node of Ti = [A,B] */
86      b0 = coordinates[bidx];
87      b1 = coordinates[bidx+nC];
88
89      for (j=0;j<nE;++j) {
90        cidx = (int) elements[j]-1;               /* 1st node of Tj = [C,D] */
91        c0 = coordinates[cidx];
92        c1 = coordinates[cidx+nC];
93
94        didx = (int) elements[j+nE]-1;            /* 2nd node of Tj = [C,D] */
95        d0 = coordinates[didx];
96        d1 = coordinates[didx+nC];
97
98        linetest1 = fabs( (a0-c0)*(b1-a1) - (a1-c1)*(b0-a0) );
99        linetest2 = fabs( (a0-d0)*(b1-a1) - (a1-d1)*(b0-a0) );
100
101
102       if( linetest1>EPS*( sqrt((a0-c0)*(a0-c0)+(a1-c1)*(a1-c1)) ) ||
103           linetest2>EPS*( sqrt((a0-d0)*(a0-d0)+(a1-d1)*(a1-d1)) ) ) {
104         /* P0xS1 - part */
105         computeKij(&I0,&I1,eta,a0,a1,b0,b1,c0,c1,d0,d1);
106         K[i+cidx*2*nE] += I0-I1;
107         K[i+didx*2*nE] += I0+I1;
108         /* P1xS1 - part */
109         J0 = computeKP1S2ij(a0,a1,b0,b1,c0,c1,d0,d1,1,0,eta);
110         J1 = computeKP1S2ij(a0,a1,b0,b1,c0,c1,d0,d1,1,1,eta);
111         K[i+nE+cidx*2*nE] += J0-J1;
112         K[i+nE+didx*2*nE] += J0+J1;
113
114         /* P0xS2 - part */
115         K[2*nE*nC + i + j*2*nE] =
116         computeKP1S2ij(a0,a1,b0,b1,c0,c1,d0,d1,0,2,eta);
117         /* P1xS2 - part */
118         K[2*nE*nC + nE + i + j*2*nE] =
119         computeKP1S2ij(a0,a1,b0,b1,c0,c1,d0,d1,1,2,eta);
120       }
121     }
122   }
123 }
```

4.2.3 computeKij

Für die Elemente $T_i = [\boldsymbol{a}, \boldsymbol{b}]$ und $T_j = [\boldsymbol{c}, \boldsymbol{d}]$ erhält die Funktion die Randpunkte $\boldsymbol{a} = [\texttt{a0}, \texttt{a1}]$, $\boldsymbol{b} = [\texttt{b0}, \texttt{b1}]$, $\boldsymbol{c} = [\texttt{c0}, \texttt{c1}]$ und $\boldsymbol{d} = [\texttt{d0}, \texttt{d1}]$. Zusätzlich wird die Zulässigkeitskonstante η übergeben. Die Funktion `computeKij` ist nun für die Realisierung des max-min-Kriteriums zuständig, welche in den Zeilen 106-120 stattfindet. Hier wurde das min-Kriterium durch das max-min-Kriterium ersetzt. Davor wird noch sichergestellt, dass T_i das kürzere Randstück ist. Dies ist für den Fall der semianalytischen Berechnung wichtig, da dort das Integral über das kürzere Randelement durch Quadratur ersetzt wird. Gegebenenfalls werden also T_i und T_j vertauscht.
Zurückgegeben wird eine Approximation für die Integrale $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vlg. (4.13)).

Listing 17: computeKij

```
80  void computeKij(double* I0, double* I1, double eta,
81                  double a0, double a1, double b0, double b1,
82                  double c0, double c1, double d0, double d1) {
83
84    int swap = 0;
85    double tmp = 0.0;
86    double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
87    double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
88
89    if(hi>hj){
90      swap = 1;                                          /* swap Tj and Ti */
91      tmp = hi;
92      hi = hj;
93      hj = tmp;
94    }
95
96    /* fully analytic computation */
97    if(eta == 0) {
98      if(swap == 0)
99        computeKijAnalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
100     else
101       computeKijSwappedAnalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
102   }
103
104   /* fully analytic or semianalytic */
105   else {
106     if(distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1)*eta >= sqrt(hj)) {
107         computeKijFullQuadrature(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
108     }
109     else if(distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1)*eta >= sqrt(hi)) {
110       if(swap == 0)
111         computeKijSemianalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
112       else
113         computeKijSwappedSemianalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
114     }
115     else {
116       if(swap == 0)
117         computeKijAnalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
118       else
119         computeKijSwappedAnalytic(I0,I1,a0,a1,b0,b1,c0,c1,d0,d1);
120     }
121   }
122 }
```

4.2.4 computeKijAnalytic

Für $\mathrm{diam}(T_i) \leq \mathrm{diam}(T_j)$ berechnet die Funktion die Integrale $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vgl. (4.13)) analytisch. Für Details zur Berechnung möchten wie auf [10, Lemma 3.1] verweisen.

Listing 18: computeKijAnalytic

```
125 void computeKijAnalytic(double* I0, double* I1,
```

```
126                              double a0, double a1, double b0, double b1,
127                              double c0, double c1, double d0, double d1) {
128
129   double lambda=0.0, mu=0.0;
130   double det=0.0;
131
132   double n[2]={0.,0.}, u[2]={0.,0.}, v[2]={0.,0.}, w[2]={0.,0.};
133   double w_plus_u[2]={0.,0.}, w_minus_u[2]={0.,0.};
134   double w_plus_v[2]={0.,0.}, w_minus_v[2]={0.,0.};
135
136   double dot_u_n=0.0, dot_w_n=0.0;
137   double dot_w_plus_u_n=0.0, dot_w_minus_u_n=0.0;
138
139   /* hi=norm2(b-a)^2 ; hj=norm2(d-c)^2 */
140   double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
141   double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
142
143   n[0] = (d1-c1)/sqrt(hj);        n[1] = (c0-d0)/sqrt(hj);     /* normaleVector */
144   u[0] = a0-b0;                   u[1] = a1-b1;                /* u=a-b         */
145   v[0] = d0-c0;                   v[1] = d1-c1;                /* v=d-c         */
146
147   w[0] = c0+d0-a0-b0;             w[1] = c1+d1-a1-b1;          /* w=c+d-a-b     */
148   w_plus_u[0] = w[0]+u[0];        w_plus_u[1] = w[1]+u[1];     /* w_plus_u=w+u  */
149   w_minus_u[0] = w[0]-u[0];       w_minus_u[1] = w[1]-u[1];    /* w_minus_u=w-u */
150
151   w_plus_v[0] = w[0]+v[0];        w_plus_v[1] = w[1]+v[1];
152   w_minus_v[0] = w[0]-v[0];       w_minus_v[1] = w[1]-v[1];
153
154   dot_u_n = u[0]*n[0]+u[1]*n[1];                                /* dot_u_n=<u,n>*/
155   dot_w_n = w[0]*n[0]+w[1]*n[1];
156   /* dot_w_plus_u_n=<w+u,n> */
157   dot_w_plus_u_n = w_plus_u[0]*n[0]+w_plus_u[1]*n[1];
158   dot_w_minus_u_n = w_minus_u[0]*n[0]+w_minus_u[1]*n[1];
159
160   det = u[0]*v[1]-u[1]*v[0];
161
162   /* u,v linearly dependent */
163   if(fabs(det)<=EPS*sqrt(hi*hj)) {
164     if(fabs(u[0])>fabs(u[1]))
165       mu = v[0]/u[0];
166     else
167       mu = v[1]/u[1];
168
169     *I0 = dot_w_n*( dlp(0,u,w_plus_v)+dlp(0,u,w_minus_v)
170     + mu*(dlp(1,v,w_minus_u)-dlp(1,v,w_plus_u)) );
171     *I1 = dot_w_n*( dlp(0,u,w_plus_v)-dlp(0,u,w_minus_v)
172     + mu*(dlp(2,v,w_minus_u)-dlp(2,v,w_plus_u)) )*0.5;
173   }
174
175   /* u,v linearly independent */
176   else{
177     if(a0==d0 && a1==d1){
178       *I0 = 2*( dot_w_plus_u_n*dlp(0,v,w_plus_u)+dot_u_n*dlp(1,u,w_minus_v)
179       + dot_w_n*dlp(0,u,w_minus_v) );
180       *I1 =     dot_w_plus_u_n*dlp(1,v,w_plus_u)-dot_u_n*dlp(1,u,w_minus_v)
181       - dot_w_n*dlp(0,u,w_minus_v) + 0.5*(*I0);
182     }
183     else if(b0==c0 && b1==c1){
184       *I0 = 2*( dot_w_minus_u_n*dlp(0,v,w_minus_u)
185       + dot_u_n*dlp(1,u,w_plus_v)+dot_w_n*dlp(0,u,w_plus_v) );
186       *I1 =     dot_w_minus_u_n*dlp(1,v,w_minus_u)
187       + dot_u_n*dlp(1,u,w_plus_v)+dot_w_n*dlp(0,u,w_plus_v)
188       - 0.5*(*I0);
189     }
190     else {
191       mu     = (w[0]*v[1]-w[1]*v[0])/det;
192       lambda = (u[0]*w[1]-u[1]*w[0])/det;
193
194       *I0 = (mu+1)*dot_w_plus_u_n*dlp(0,v,w_plus_u)
195       - (mu-1)*dot_w_minus_u_n*dlp(0,v,w_minus_u)
196       + (lambda+1)*(dot_u_n*dlp(1,u,w_plus_v)
197       + dot_w_n*dlp(0,u,w_plus_v))
198       - (lambda-1)*(dot_u_n*dlp(1,u,w_minus_v)
199       + dot_w_n*dlp(0,u,w_minus_v));
```

```
200        *I1 = 0.5*( (mu+1)*dot_w_plus_u_n*dlp(1,v,w_plus_u)
201        - (mu-1)*dot_w_minus_u_n*dlp(1,v,w_minus_u)
202        + (lambda+1)*(dot_u_n*dlp(1,u,w_plus_v)
203        + dot_w_n*dlp(0,u,w_plus_v))
204        + (lambda-1)*(dot_u_n*dlp(1,u,w_minus_v)
205        + dot_w_n*dlp(0,u,w_minus_v))
206        - lambda*(*I0) );
207      }
208    }
209    *I0 *= -0.125*sqrt(hi*hj)/M_PI;
210    *I1 *= -0.125*sqrt(hi*hj)/M_PI;
211  }
```

4.2.5 computeKijSwappedAnalytic

Für den Fall, dass $\mathrm{diam}(T_i) > \mathrm{diam}(T_j)$ werden die beiden Integrale bei jeweils $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vgl. (4.13)) so vertauscht, dass das innere Integral dem Integral über T_i entspricht. Mit anderen Worten: Es soll wieder zuerst über das längere Randstück, in diesem Fall T_i, und danach über das kürzere Randstück T_j integriert werden. Genauer gesagt werden hier die Integrale

$$-\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(s))\cdot n(\gamma_j(t))}{|\gamma_i(s)-\gamma_j(t)|^2}\frac{1}{2}\,ds\,dt,$$
$$-\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi}\int_{-1}^{1}\int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(s))\cdot n(\gamma_j(t))}{|\gamma_i(s)-\gamma_j(t)|^2}\frac{t}{2}\,ds\,dt$$

berechnet. Diese zwei Doppelintegrale werden nun mit Hilfe von [10, Lemma 3.1] anders als bei `computeKijAnalytic` berechnet. Auch hier möchten wir nicht auf die genaue Berechnung eingehen.

Listing 19: computeKijSwappedAnalytic

```
214  void computeKijSwappedAnalytic(double* I0, double* I1,
215                                 double a0, double a1, double b0, double b1,
216                                 double c0, double c1, double d0, double d1) {
217
218    double lambda = 0;
219    double mu = 0;
220    double det = 0;
221
222    /*hi=norm2(b-a)^2 ; hj=norm2(d-c)^2 */
223    double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
224    double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
225
226    double n[2], u[2], v[2], w[2];
227    double w_plus_u[2], w_minus_u[2], w_plus_v[2], w_minus_v[2];
228    double dot_v_n, dot_w_n, dot_w_plus_v_n, dot_w_minus_v_n;
229
230    n[0] = (d1-c1)/sqrt(hj);        n[1] = (c0-d0)/sqrt(hj);   /* normaleVector */
231    u[0] = d0-c0;                   u[1] = d1-c1;              /* u=d-c         */
232    v[0] = a0-b0;                   v[1] = a1-b1;              /* v=a-b         */
233
234    w[0] = c0+d0-a0-b0;             w[1] = c1+d1-a1-b1;        /* w=c+d-a-b     */
235    w_plus_u[0] = w[0]+u[0];        w_plus_u[1] = w[1]+u[1];   /* w_plus_u=w+u  */
236    w_minus_u[0] = w[0]-u[0];       w_minus_u[1] = w[1]-u[1];  /* w_minus_u=w-u */
237
238    w_plus_v[0] = w[0]+v[0];        w_plus_v[1] = w[1]+v[1];
239    w_minus_v[0] = w[0]-v[0];       w_minus_v[1] = w[1]-v[1];
240
241    dot_v_n = v[0]*n[0]+v[1]*n[1];                              /* dot_u_n=<u,n>*/
242    dot_w_n = w[0]*n[0]+w[1]*n[1];
243    /* dot_w_plus_u_n=<w+u,n> */
244    dot_w_plus_v_n = w_plus_v[0]*n[0]+w_plus_v[1]*n[1];
245    dot_w_minus_v_n = w_minus_v[0]*n[0]+w_minus_v[1]*n[1];
```

```
246
247   det = u[0]*v[1]-u[1]*v[0];
248
249   /* u,v linearly dependent */
250   if(fabs(det)<=EPS*sqrt(hi*hj)) {
251     if(fabs(u[0])>fabs(u[1]))
252       mu = v[0]/u[0];
253     else
254       mu = v[1]/u[1];
255
256     *I0 = dot_w_n*( dlp(0,u,w_plus_v)+dlp(0,u,w_minus_v)
257     + mu*(dlp(1,v,w_minus_u)-dlp(1,v,w_plus_u)) );
258     *I1 = dot_w_n*( dlp(1,u,w_plus_v)+dlp(1,u,w_minus_v)
259     + mu*( -dlp(1,v,w_minus_u)-dlp(1,v,w_plus_u)
260     +0.5*( dlp(0,u,w_plus_v)-dlp(0,u,w_minus_v)
261     + mu*(dlp(2,v,w_minus_u)-dlp(2,v,w_plus_u)))));
262   }
263
264   /* u,v linearly independent */
265   else{
266     if(a0==d0 && a1==d1){
267       *I0 = 2*( dot_v_n*dlp(1,v,w_minus_u)+dot_w_n*dlp(0,v,w_minus_u)
268       + dot_w_plus_v_n*dlp(0,u,w_plus_v) );
269       *I1 =      dot_w_plus_v_n*dlp(1,u,w_plus_v)-dot_v_n*dlp(1,v,w_minus_u)
270       - dot_w_n*dlp(0,v,w_minus_u) + 0.5*(*I0);
271     }
272     else if(b0==c0 && b1==c1){
273       *I0 = 2*( dot_v_n*dlp(1,v,w_plus_u)
274       +dot_w_n*dlp(0,v,w_plus_u)+dot_w_minus_v_n*dlp(0,u,w_minus_v) );
275       *I1 =      dot_v_n*dlp(1,v,w_plus_u)+dot_w_n*dlp(0,v,w_plus_u)
276       + dot_w_minus_v_n*dlp(1,u,w_minus_v) - 0.5*(*I0);
277     }
278     else {
279       mu     = (w[0]*v[1]-w[1]*v[0])/det;
280       lambda = (u[0]*w[1]-u[1]*w[0])/det;
281
282       *I0 = (mu+1)*(dot_v_n*dlp(1,v,w_plus_u)+dot_w_n*dlp(0,v,w_plus_u))
283       - (mu-1)*(dot_v_n*dlp(1,v,w_minus_u)+dot_w_n*dlp(0,v,w_minus_u))
284       + (lambda+1)*dot_w_plus_v_n*dlp(0,u,w_plus_v)
285       - (lambda-1)*dot_w_minus_v_n*dlp(0,u,w_minus_v);
286       *I1 = 0.5*( (mu+1)*(dot_v_n*dlp(1,v,w_plus_u)+dot_w_n*dlp(0,v,w_plus_u))
287       +(mu-1)*(dot_v_n*dlp(1,v,w_minus_u)+dot_w_n*dlp(0,v,w_minus_u))
288       +(lambda+1)*dot_w_plus_v_n*dlp(1,u,w_plus_v)
289       -(lambda-1)*dot_w_minus_v_n*dlp(1,u,w_minus_v)-mu*(*I0));
290     }
291   }
292   *I0 *= -0.125*sqrt(hi*hj)/M_PI;
293   *I1 *= -0.125*sqrt(hi*hj)/M_PI;
294 }
```

4.2.6 computeKijSemianalytic

Diese Funktion berechnet nun die Doppelintegrale $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vgl. (4.13)) semianalytisch. Dabei wird das Integral über das kürzere Randstück, in unserem Fall T_i, durch Quadratur ersetzt. Es werden also

$$I^0_{0,i,j} \simeq -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi}\sum_{k=0}^{p}\omega_k\int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(z_k))\cdot n(\gamma_j(t))}{|\gamma_j(t)-\gamma_i(z_k)|^2}\frac{1}{2}\,dt,$$

$$I^0_{1,i,j} \simeq -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi}\sum_{k=0}^{p}\omega_k\int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(z_k))\cdot n(\gamma_j(t))}{|\gamma_j(t)-\gamma_i(z_k)|^2}\frac{t}{2}\,dt$$

approximiert. Für unsere Berechnung ist der Quadraturgrad $p =$ `GAUSS_ORDER` $- 1$ aus (4.4).

Listing 20: computeKijSemianalytic

```
297 void computeKijSemianalytic(double* I0, double* I1,
298                             double a0, double a1, double b0, double b1,
299                             double c0, double c1, double d0, double d1) {
300
301   /*hi=norm2(b-a)^2 hj=norm2(d-c)^2 */
302   double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
303   double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
304
305   int i=0;
306   double n[2], u[2], v[2], w[2], z[2];
307   double I0tmp = 0.0;
308   double I1tmp = 0.0;
309   double dot_z_n = 0.0;
310
311   /* 16-point Gaussian quadrature on [-1,1] */
312   const int order = 16;
313   const double* gauss_weight = getGaussWeights(order);
314   const double* gauss_point  = getGaussPoints(order);
315
316   n[0] = (d1-c1)/sqrt(hj);       n[1] = (c0-d0)/sqrt(hj);        /* normaleVector */
317   u[0] = a0-b0;                  u[1] = a1-b1;                   /* u=a-b         */
318   v[0] = d0-c0;                  v[1] = d1-c1;                   /* v=d-c         */
319   w[0] = c0+d0-a0-b0;            w[1] = c1+d1-a1-b1;             /* w=c+d-a-b     */
320
321   for (i=0;i<order;++i) {
322     z[0] = gauss_point[i]*u[0]+w[0];
323     z[1] = gauss_point[i]*u[1]+w[1];
324
325     dot_z_n = z[0]*n[0]+z[1]*n[1];
326
327     I0tmp += gauss_weight[i]*dot_z_n*dlp(0,v,z);
328     I1tmp += gauss_weight[i]*dot_z_n*dlp(1,v,z);
329   }
330
331   *I0 = -0.125*sqrt(hi*hj)*I0tmp/M_PI;
332   *I1 = -0.125*sqrt(hi*hj)*I1tmp/M_PI;
333 }
```

4.2.7 `computeKijSwappedSemianalytic`

Falls nun $\mathrm{diam}(T_i) > \mathrm{diam}(T_j)$, werden die beiden Integrale bei $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vgl. (4.13)) so vertauscht, dass das innere Integral dem Integral über T_i entspricht. Dann wird zuerst das Integral über T_i exakt berechnet und anschließend das äußere Integral über das kürzere Randstück T_j mittels Quadratur approximiert. Im Gegensatz zu `computeKijSemianalytic` sind die Approximationen nun leicht abgeändert:

$$I^0_{0,i,j} \simeq -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi} \sum_{k=0}^{p} \omega_k \int_{-1}^{1} \frac{(\gamma_j(z_k) - \gamma_i(s)) \cdot n(\gamma_j(z_k))}{|\gamma_j(z_k) - \gamma_i(s)|^2} \frac{1}{2}\, ds,$$

$$I^0_{1,i,j} \simeq -\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{8\pi} \sum_{k=0}^{p} \omega_k \int_{-1}^{1} \frac{(\gamma_j(z_k) - \gamma_i(s)) \cdot n(\gamma_j(z_k))}{|\gamma_j(z_k) - \gamma_i(s)|^2} \frac{z_k}{2}\, ds.$$

Dabei ist $p =$ `GAUSS_ORDER` $- 1$ aus (4.4).

Listing 21: computeKijSwappedSemianalytic

```
335 void computeKijSwappedSemianalytic(double* I0, double* I1,
336                                    double a0, double a1, double b0, double b1,
337                                    double c0, double c1, double d0, double d1) {
338
339   /*hi=norm2(b-a)^2 ; hj=norm2(d-c)^2 */
340   double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
341   double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
342
343   int i=0;
344   double n[2], u[2], v[2], w[2], z[2];
```

```
345     double dot_v_n , dot_w_n;
346     double I0tmp = 0.0;
347     double I1tmp = 0.0;
348
349     /* 16-point Gaussian quadrature on [-1,1] */
350     const int order = 16;
351     const double* gauss_weight = getGaussWeights(order);
352     const double* gauss_point  = getGaussPoints(order);
353
354
355     n[0] = (d1-c1)/sqrt(hj);      n[1] = (c0-d0)/sqrt(hj);       /* normaleVector */
356     u[0] = d0-c0;                 u[1] = d1-c1;                  /* u=d-c         */
357     v[0] = a0-b0;                 v[1] = a1-b1;                  /* v=a-b         */
358     w[0] = c0+d0-a0-b0;           w[1] = c1+d1-a1-b1;            /* w=c+d-a-b     */
359
360     dot_v_n = v[0]*n[0]+v[1]*n[1];
361     dot_w_n = w[0]*n[0]+w[1]*n[1];
362
363     for (i=0;i<order;++i) {
364       z[0] = gauss_point[i]*u[0]+w[0];
365       z[1] = gauss_point[i]*u[1]+w[1];
366
367       I0tmp += gauss_weight[i]*( dot_v_n*dlp(1,v,z) + dot_w_n*dlp(0,v,z) );
368       I1tmp += gauss_weight[i]*gauss_point[i]*( dot_v_n*dlp(1,v,z)
369       + dot_w_n*dlp(0,v,z) );
370     }
371
372     *I0 = -0.125*sqrt(hi*hj)*I0tmp/M_PI;
373     *I1 = -0.125*sqrt(hi*hj)*I1tmp/M_PI;
374   }
```

4.2.8 computeKijFullQuadrature

Diese Funktion gibt nun eine approximative Berechnung von $I^0_{0,i,j}$ und $I^0_{1,i,j}$ (vgl. (4.13)) zurück, wobei nun beide Integrale mit Quadratur angenähert werden. Für $p = \texttt{GAUSS_ORDER} - 1$ aus (4.4) gilt:

$$I^0_{0,i,j} \simeq -\frac{\text{diam}(T_i)\text{diam}(T_j)}{8\pi} \sum_{k=0}^{p} \sum_{m=0}^{p} \omega_k \omega_m \frac{(\gamma_j(z_m) - \gamma_i(z_k)) \cdot n(\gamma_j(z_m))}{|\gamma_j(z_m) - \gamma_i(z_k)|^2} \frac{1}{2},$$

$$I^0_{1,i,j} \simeq -\frac{\text{diam}(T_i)\text{diam}(T_j)}{8\pi} \sum_{k=0}^{p} \sum_{m=0}^{p} \omega_k \omega_m \frac{(\gamma_j(z_m) - \gamma_i(z_k)) \cdot n(\gamma_j(z_m))}{|\gamma_j(z_m) - \gamma_i(z_k)|^2} \frac{z_m}{2}.$$

Listing 22: computeKijFullQuadrature

```
806   void computeKijFullQuadrature(double* I0, double* I1,
807                                 double a0, double a1, double b0, double b1,
808                                 double c0, double c1, double d0, double d1) {
809
810     /*hi=norm2(b-a)^2 hj=norm2(d-c)^2 */
811     double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
812     double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
813
814     int i=0,k=0;
815     double n[2], u[2], v[2], w[2], z[2], r[2];
816     double I0tmp = 0.0;
817     double I1tmp = 0.0;
818     double dot_z_n = 0.0;
819
820     /* 16-point Gaussian quadrature on [-1,1] */
821     const int order = 16;
822     const double* gauss_weight = getGaussWeights(order);
823     const double* gauss_point  = getGaussPoints(order);
824
825     n[0] = (d1-c1)/sqrt(hj);      n[1] = (c0-d0)/sqrt(hj);       /* normaleVector */
826     u[0] = a0-b0;                 u[1] = a1-b1;                  /* u=a-b         */
```

```
827   v[0] = d0-c0;               v[1] = d1-c1;                /* v=d-c        */
828   w[0] = c0+d0-a0-b0;         w[1] = c1+d1-a1-b1;          /* w=c+d-a-b    */
829
830   for (i=0;i<order;++i) {
831     z[0] = gauss_point[i]*u[0]+w[0];
832     z[1] = gauss_point[i]*u[1]+w[1];
833
834     dot_z_n = z[0]*n[0]+z[1]*n[1];
835
836     for (k=0;k<order;++k) {
837         r[0] = w[0]+gauss_point[k]*v[0]+gauss_point[i]*u[0];
838         r[1] = w[1]+gauss_point[k]*v[1]+gauss_point[i]*u[1];
839
840         I0tmp += gauss_weight[i]*gauss_weight[k]*dot_z_n/(r[0]*r[0]+r[1]*r[1]);
841         I1tmp += gauss_weight[i]*gauss_weight[k]*gauss_point[k]*
842                 dot_z_n/(r[0]*r[0]+r[1]*r[1]);
843     }
844   }
845
846   *I0 = -0.125*sqrt(hi*hj)*I0tmp/M_PI;
847   *I1 = -0.125*sqrt(hi*hj)*I1tmp/M_PI;
848 }
```

4.2.9 computeKP1S2ij

Die Funktion erhält die Elemente $T_i = [\boldsymbol{a}, \boldsymbol{b}], T_j = [\boldsymbol{c}, \boldsymbol{d}]$ in Form ihrer Randpunkte $\boldsymbol{a} = [\texttt{a0}, \texttt{a1}]$, $\boldsymbol{b} = [\texttt{b0}, \texttt{b1}]$, $\boldsymbol{c} = [\texttt{c0}, \texttt{c1}]$ und $\boldsymbol{d} = [\texttt{d0}, \texttt{d1}]$. Zusätzlich wird die Zulässigkeitskonstante η übergeben. Darüber hinaus werden auch noch Werte $\texttt{tfun} = \ell \in \{0, 1\}$ und $\texttt{afun} = m \in \{1, 2\}$ übergeben. Die benötigten Doppelintegrale werden nun mit Hilfe des max-min-Kriteriums, welches in den Zeilen 620-628 realisiert ist, berechnet. Dabei wird sichergestellt, dass $\operatorname{diam}(T_i) \leq \operatorname{diam}(T_j)$. Andernfalls werden die beiden Elemente vertauscht.
In Abhängigkeit der übergebenen Werte ℓ, m werden nun unterschiedliche Integrale berechnet. Falls $m = 2$ und $\ell \in \{0, 1\}$ wird das Integral

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{32\pi} \int_{-1}^{1} \int_{-1}^{1} \frac{(\gamma_j(t) - \gamma_i(s)) \cdot n(\gamma_j(t))}{|\gamma_i(s) - \gamma_j(t)|^2} (1 - t^2) s^\ell \, dt \, ds \tag{4.16}$$

berechnet. Für $m \in \{0, 1\}$ und $\ell = 1$ wird das Integral

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{16\pi} \int_{-1}^{1} \int_{-1}^{1} \frac{(\gamma_j(t) - \gamma_i(s)) \cdot n(\gamma_j(t))}{|\gamma_i(s) - \gamma_j(t)|^2} t^m s \, dt \, ds \tag{4.17}$$

berechnet.

Listing 23: computeKP1S2ij

```
595 double computeKP1S2ij(double a0, double a1, double b0, double b1,
596                 double c0, double c1, double d0, double d1, int tfun, int afun,
597                 double eta) {
598
599   int swap = 0;
600   double tmp = 0.0;
601   double hi = (b0-a0)*(b0-a0)+(b1-a1)*(b1-a1);
602   double hj = (d0-c0)*(d0-c0)+(d1-c1)*(d1-c1);
603
604   double val = 0.0;
605
606   if(hi>hj){
607     swap = 1;
608     tmp = hi;
609     hi = hj;
610     hj = tmp;
```

```
611     }
612
613     /* fully analytic computation */
614     if(eta == 0) {
615       return computeKP1S2ijAnalytic(a0,a1,b0,b1,c0,c1,d0,d1,tfun,afun,swap);
616     }
617
618     /* fully analytic or semianalytic */
619     else {
620       if(distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1)*eta >= sqrt(hj)) {
621         return computeKP1S2ijFullQuadrature(a0,a1,b0,b1,c0,c1,d0,d1,tfun,afun,swap);
622       }
623       else if(distanceSegmentToSegment(a0,a1,b0,b1,c0,c1,d0,d1)*eta >= sqrt(hi)) {
624         return computeKP1S2ijSemianalytic(a0,a1,b0,b1,c0,c1,d0,d1,tfun,afun,swap);
625       }
626       else {
627         return computeKP1S2ijAnalytic(a0,a1,b0,b1,c0,c1,d0,d1,tfun,afun,swap);
628       }
629     }
630   }
```

4.2.10 computeKP1S2ijAnalytic

Die Funktion berechnet die Integrale (4.16) für $m = 2$ und $\ell \in \{0, 1\}$ und (4.17) für $m \in \{0, 1\}$ und $\ell = 1$ exakt. Falls $\text{diam}(T_i) > \text{diam}(T_j)$ wird der wesentliche Teil der Integralberechnung über die Funktion doubleDlpSwapped realisiert. Andernfalls wird die Funktion doubleDlp verwendet um die Integrale zu berechnen.

Listing 24: computeKP1S2ijAnalytic

```
746   double computeKP1S2ijAnalytic(double a0, double a1, double b0, double b1,
747     double c0, double c1, double d0, double d1, int tfun, int afun, int swap) {
748
749     double val = 0.0;
750
751     int i=0;
752     int j=0;
753     int A_eq_D = 0, B_eq_C = 0;
754
755     double u[2] = {0.,0.}, v[2] = {0.,0.}, w[2] = {0.,0.};
756
757     double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1);
758     double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1);
759
760     u[0] = a0-b0; u[1] = a1-b1;
761     v[0] = d0-c0; v[1] = d1-c1;
762     w[0] = c0+d0-a0-b0; w[1] = c1+d1-a1-b1;
763
764     /* Check if A==D || B == C */
765     if(a0 == d0 && a1 == d1)
766       A_eq_D = 1;
767     if(b0 == c0 && b1 == c1)
768       B_eq_C = 1;
769
770     /* Check if segments were swapped */
771     if(swap) {
772       if( tfun==0 && afun==2) {
773         val = -0.0625/M_PI * sqrt(hi*hj) *
774         (doubleDlpSwapped(0,0,v,u,w,A_eq_D,B_eq_C) -
775         doubleDlpSwapped(2,0,v,u,w,A_eq_D,B_eq_C) );
776       }
777       else if( tfun==1 && afun==2) {
778         val = -0.0625/M_PI * sqrt(hi*hj) *
779         (doubleDlpSwapped(0,1,v,u,w,A_eq_D,B_eq_C) -
780         doubleDlpSwapped(2,1,v,u,w,A_eq_D,B_eq_C) );
781       }
782       else if( tfun == 1 && afun == 0)
783         val = -0.125/M_PI * sqrt(hi*hj) * doubleDlpSwapped(0,1,v,u,w,A_eq_D,B_eq_C);
784       else if (tfun ==1 && afun == 1)
785         val = -0.125/M_PI * sqrt(hi*hj) * doubleDlpSwapped(1,1,v,u,w,A_eq_D,B_eq_C);
```

```
786
787   }
788   else {
789     if( tfun==0 && afun==2 ) {
790       val = -0.0625/M_PI *sqrt(hi*hj) * (doubleDlp(0,0,u,v,w,A_eq_D,B_eq_C) -
791       doubleDlp(0,2,u,v,w,A_eq_D,B_eq_C) );
792     }
793     else if(tfun ==1 && afun==2) {
794       val = -0.0625/M_PI * sqrt(hi*hj) * (doubleDlp(1,0,u,v,w,A_eq_D,B_eq_C)-
795       doubleDlp(1,2,u,v,w,A_eq_D,B_eq_C) );
796     }
797     else if( tfun == 1 && afun == 0)
798       val = -0.125/M_PI * sqrt(hi*hj) * doubleDlp(1,0,u,v,w,A_eq_D,B_eq_C);
799     else if (tfun ==1 && afun == 1)
800       val = -0.125/M_PI * sqrt(hi*hj) * doubleDlp(1,1,u,v,w,A_eq_D,B_eq_C);
801   }
802   return val;
803 }
```

4.2.11 computeKP1S2ijSemianalytic

Die Funktion berechnet die Integrale (4.16) für $m = 2$ und $\ell \in \{0,1\}$ und (4.17) für $m \in \{0,1\}$ und $\ell = 1$ semianalytisch. An dieser Stelle möchten wir nochmals hervorheben, dass in beiden Fällen das innere Integral dem Integral über T_j und das äußere dem Integral über T_i entspricht. Die Darstellungen (4.16) und (4.17) sind dann durch Transformation entstanden. Für den Fall $\operatorname{diam}(T_i) \le \operatorname{diam}(T_j)$ werden die inneren Integrale über T_j exakt berechnet und die äußeren Integrale über das kürzere Randstück T_i mittels Quadratur approximiert. Für $m = 2$ und $\ell \in \{0,1\}$ wird also

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{32\pi}\sum_{k=0}^{p}\omega_k z_k^\ell \int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(z_k))\cdot n(\gamma_j(t))}{|\gamma_i(z_k)-\gamma_j(t)|^2}(1-t^2)\,dt$$

berechnet. Für $m \in \{0,1\}$ und $\ell = 1$ wird

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{16\pi}\sum_{k=0}^{p}\omega_k z_k \int_{-1}^{1}\frac{(\gamma_j(t)-\gamma_i(z_k))\cdot n(\gamma_j(t))}{|\gamma_i(z_k)-\gamma_j(t)|^2}t^m\,dt$$

berechnet. Für den Fall, dass $\operatorname{diam}(T_i) > \operatorname{diam}(T_j)$ werden die Integrale vertauscht und das innere Integral über T_i wird nun exakt berechnet wohingegen das äußere Integral über T_j durch Quadratur ersetzt wird. Für $m = 2$ und $\ell \in \{0,1\}$ wird also

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{32\pi}\sum_{k=0}^{p}\omega_k(1-z_k^2)\int_{-1}^{1}\frac{(\gamma_j(z_k)-\gamma_i(s))\cdot n(\gamma_j(z_k))}{|\gamma_i(s)-\gamma_j(z_k)|^2}s^\ell\,ds$$

berechnet. Für $m \in \{0,1\}$ und $\ell = 1$ wird

$$-\frac{\operatorname{diam}(T_i)\operatorname{diam}(T_j)}{16\pi}\sum_{k=0}^{p}\omega_k z_k^m \int_{-1}^{1}\frac{(\gamma_j(z_k)-\gamma_i(s))\cdot n(\gamma_j(z_k))}{|\gamma_i(s)-\gamma_j(z_k)|^2}s\,ds$$

berechnet. In allen Fällen gilt $p = \texttt{GAUSS_ORDER} - 1$.

Listing 25: computeKP1S2ijSemianalytic

```
632 double computeKP1S2ijSemianalytic(double a0, double a1, double b0, double b1,
633         double c0,  double c1, double d0, double d1, int tfun, int afun, int swap) {
634
635   double val = 0.0;
```

```
636
637   double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1);
638   double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1);
639
640   int i = 0;
641   double n[2], u[2], v[2], w[2], z[2];
642   double dot_z_n=0.0;
643   double dot_u_n=0.0;
644   double dot_w_n=0.0;
645
646   /* 16 point Gauss quadr on [-1,1] */
647   const int order = 16;
648   const double* gauss_weight = getGaussWeights(order);
649   const double* gauss_point = getGaussPoints(order);
650
651   n[0] = (d1-c1)/sqrt(hj); n[1] = (c0-d0)/sqrt(hj);
652   u[0] = a0-b0; u[1] = a1-b1;
653   v[0] = d0-c0; v[1] = d1-c1;
654   w[0] = c0+d0-a0-b0; w[1] = c1+d1-a1-b1;
655
656   dot_u_n = u[0]*n[0]+u[1]*n[1];
657   dot_w_n = w[0]*n[0]+w[1]*n[1];
658
659   if(swap) { /* Aufpassen! hier werden u und v getauscht!!*/
660     if(tfun==0 && afun ==2) {
661      for(i=0; i<order; ++i) {
662         z[0] = gauss_point[i]*v[0]+w[0];
663         z[1] = gauss_point[i]*v[1]+w[1];
664
665         val += gauss_weight[i]*(1-gauss_point[i]*gauss_point[i]) *
666         (dot_u_n*dlp(1,u,z) + dot_w_n*dlp(0,u,z));
667       }
668     val = -0.0625/M_PI*sqrt(hi*hj)*val;
669     }
670     else if(tfun==1 && afun ==2) {
671      for(i=0; i<order; ++i) {
672         z[0] = gauss_point[i]*v[0]+w[0];
673             z[1] = gauss_point[i]*v[1]+w[1];
674
675         val += gauss_weight[i]*(1-gauss_point[i]*gauss_point[i]) *
676         (dot_u_n*dlp(2,u,z) + dot_w_n*dlp(1,u,z));
677       }
678     val = -0.0625/M_PI*sqrt(hi*hj)*val;
679     }
680    else if(tfun==1 && afun ==0) {
681      for(i=0; i<order; ++i) {
682         z[0] = gauss_point[i]*v[0]+w[0];
683             z[1] = gauss_point[i]*v[1]+w[1];
684
685         val += gauss_weight[i]*
686         (dot_u_n*dlp(2,u,z) + dot_w_n*dlp(1,u,z));
687       }
688     val = -0.125/M_PI*sqrt(hi*hj)*val;
689     }
690    else if(tfun==1 && afun ==1) {
691      for(i=0; i<order; ++i) {
692         z[0] = gauss_point[i]*v[0]+w[0];
693             z[1] = gauss_point[i]*v[1]+w[1];
694
695         val += gauss_weight[i]*gauss_point[i]*
696         (dot_u_n*dlp(2,u,z) + dot_w_n*dlp(1,u,z));
697       }
698     val = -0.125/M_PI*sqrt(hi*hj)*val;
699     }
700   }
701   else {
702     if(tfun==0 && afun ==2) {
703       for(i=0; i<order; ++i) {
704         z[0] = gauss_point[i]*u[0]+w[0];
705         z[1] = gauss_point[i]*u[1]+w[1];
706         dot_z_n = z[0]*n[0]+z[1]*n[1];
707
708         val += gauss_weight[i]*dot_z_n*(dlp(0,v,z)-dlp(2,v,z));
709       }
```

```
710         val = -1/(16*M_PI)*sqrt(hi*hj)*val;
711       }
712       else if(tfun==1 && afun==2) {
713         for(i=0; i<order; ++i) {
714           z[0] = gauss_point[i]*u[0]+w[0];
715           z[1] = gauss_point[i]*u[1]+w[1];
716           dot_z_n = z[0]*n[0]+z[1]*n[1];
717
718           val += gauss_weight[i]*gauss_point[i]*dot_z_n*( dlp(0,v,z)-dlp(2,v,z));
719         }
720         val = -1/(16*M_PI)*sqrt(hi*hj)*val;
721       }
722       else if(tfun==1 && afun ==0) {
723          for(i=0; i<order; ++i) {
724           z[0] = gauss_point[i]*u[0]+w[0];
725           z[1] = gauss_point[i]*u[1]+w[1];
726           dot_z_n = z[0]*n[0]+z[1]*n[1];
727
728           val += gauss_weight[i]*gauss_point[i]*dot_z_n* dlp(0,v,z);
729         }
730         val = -1/(8*M_PI)*sqrt(hi*hj)*val;
731       }
732       else if(tfun==1 && afun ==1) {
733          for(i=0; i<order; ++i) {
734           z[0] = gauss_point[i]*u[0]+w[0];
735           z[1] = gauss_point[i]*u[1]+w[1];
736           dot_z_n = z[0]*n[0]+z[1]*n[1];
737
738           val += gauss_weight[i]*gauss_point[i]*dot_z_n* dlp(1,v,z);
739         }
740         val = -1/(8*M_PI)*sqrt(hi*hj)*val;
741       }
742     }
743    return val;
744  }
```

4.2.12 computeKP1S2ijFullQuadrature

Die Funktion berechnet die Integrale (4.16) für $m = 2$ und $\ell \in \{0, 1\}$ und (4.17) für $m \in \{0, 1\}$ und $\ell = 1$, wobei alle Integrale mittels Gauß-Quadratur approximiert werden. Für $m = 2$ und $\ell \in \{0, 1\}$ wird also

$$-\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{32\pi}\sum_{k=0}^{p}\sum_{h=0}^{p}\omega_k\omega_h z_k^{\ell}(1-z_h^2)\frac{(\gamma_j(z_h)-\gamma_i(z_k))\cdot n(\gamma_j(z_h))}{|\gamma_i(z_k)-\gamma_j(z_h)|^2}$$

berechnet. Für $m \in \{0, 1\}$ und $\ell = 1$ wird

$$-\frac{\mathrm{diam}(T_i)\mathrm{diam}(T_j)}{16\pi}\sum_{k=0}^{p}\sum_{h=0}^{p}\omega_k\omega_h z_k z_h^m\frac{(\gamma_j(z_h)-\gamma_i(z_k))\cdot n(\gamma_j(z_h))}{|\gamma_i(z_k)-\gamma_j(z_h)|^2}$$

berechnet. Dabei gilt $p = \texttt{GAUSS_ORDER} - 1$.

Listing 26: computeKP1S2ijFullQuadrature

```
900  double computeKP1S2ijFullQuadrature(double a0, double a1, double b0,
901          double b1, double c0,  double c1, double d0, double d1, int tfun,
902          int afun, int swap) {
903    /* tfun - Ordnung der P-Funktionen */
904    /* afun - Ordnung der S-Funktionen */
905
906    double val = 0.0;
907
908    double hi = (b0-a0)*(b0-a0) + (b1-a1)*(b1-a1);
909    double hj = (d0-c0)*(d0-c0) + (d1-c1)*(d1-c1);
910
```

```
911     int i = 0,k = 0;
912     double n[2], u[2], v[2], w[2], z[2], r[2];
913     double dot_z_n=0.0;
914     double dot_u_n=0.0;
915     double dot_w_n=0.0;
916
917     /* 16 point Gauss quadr on [-1,1] */
918     const int order = 16;
919     const double* gauss_weight = getGaussWeights(order);
920     const double* gauss_point = getGaussPoints(order);
921
922     n[0] = (d1-c1)/sqrt(hj); n[1] = (c0-d0)/sqrt(hj);
923     u[0] = a0-b0; u[1] = a1-b1;
924     v[0] = d0-c0; v[1] = d1-c1;
925     w[0] = c0+d0-a0-b0; w[1] = c1+d1-a1-b1;
926
927     if(tfun==0 && afun ==2) {
928         for(i=0; i<order; ++i) {
929           z[0] = gauss_point[i]*u[0]+w[0];
930           z[1] = gauss_point[i]*u[1]+w[1];
931           dot_z_n = z[0]*n[0]+z[1]*n[1];
932
933           for(k=0; k<order; ++k) {
934               r[0] = w[0]+gauss_point[k]*v[0]+gauss_point[i]*u[0];
935               r[1] = w[1]+gauss_point[k]*v[1]+gauss_point[i]*u[1];
936
937               val += gauss_weight[i]*gauss_weight[k]*dot_z_n*
938                       (1-gauss_point[k]*gauss_point[k])/(r[0]*r[0]+r[1]*r[1]);
939           }
940         }
941         val = -1/(16*M_PI)*sqrt(hi*hj)*val;
942     }
943     else if(tfun==1 && afun==2) {
944        for(i=0; i<order; ++i) {
945           z[0] = gauss_point[i]*u[0]+w[0];
946           z[1] = gauss_point[i]*u[1]+w[1];
947           dot_z_n = z[0]*n[0]+z[1]*n[1];
948
949           for(k=0; k<order; ++k) {
950               r[0] = w[0]+gauss_point[k]*v[0]+gauss_point[i]*u[0];
951               r[1] = w[1]+gauss_point[k]*v[1]+gauss_point[i]*u[1];
952
953               val += gauss_weight[i]*gauss_point[i]*gauss_weight[k]*dot_z_n*
954                       (1-gauss_point[k]*gauss_point[k])/(r[0]*r[0]+r[1]*r[1]);
955           }
956         }
957         val = -1/(16*M_PI)*sqrt(hi*hj)*val;
958     }
959     else if(tfun==1 && afun ==0) {
960        for(i=0; i<order; ++i) {
961           z[0] = gauss_point[i]*u[0]+w[0];
962           z[1] = gauss_point[i]*u[1]+w[1];
963           dot_z_n = z[0]*n[0]+z[1]*n[1];
964
965           for (k=0;k<order;++k) {
966               r[0] = w[0]+gauss_point[k]*v[0]+gauss_point[i]*u[0];
967               r[1] = w[1]+gauss_point[k]*v[1]+gauss_point[i]*u[1];
968
969               val += gauss_weight[i]*gauss_weight[k]*gauss_point[i]*dot_z_n/
970                       (r[0]*r[0]+r[1]*r[1]);
971           }
972         }
973         val = -1/(8*M_PI)*sqrt(hi*hj)*val;
974     }
975     else if(tfun==1 && afun ==1) {
976        for(i=0; i<order; ++i) {
977           z[0] = gauss_point[i]*u[0]+w[0];
978           z[1] = gauss_point[i]*u[1]+w[1];
979           dot_z_n = z[0]*n[0]+z[1]*n[1];
980
981           for (k=0;k<order;++k) {
982               r[0] = w[0]+gauss_point[k]*v[0]+gauss_point[i]*u[0];
983               r[1] = w[1]+gauss_point[k]*v[1]+gauss_point[i]*u[1];
984
```

```
985                 val += gauss_weight[i]*gauss_weight[k]*gauss_point[i]*
986                        gauss_point[k]*dot_z_n/(r[0]*r[0]+r[1]*r[1]);
987             }
988           }
989           val = -1/(8*M_PI)*sqrt(hi*hj)*val;
990         }
991    return val;
992  }
```

4.2.13 dlp

Die Funktion berechnet für $k \in \mathbb{N}_0$ und $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^2$ das Integral

$$\int_{-1}^{1} \frac{t^k}{|t\boldsymbol{p} + \boldsymbol{q}|}\, dt.$$

Für Details zur Berechnung möchten wir auf [10, Lemma 2.1] verweisen.

Listing 27: dlp

```
23  double dlp(int k, double* p, double* q) {
24    double a = p[0]*p[0]+p[1]*p[1];                    /* a = |p|^2   */
25    double b = 2*( p[0]*q[0]+p[1]*q[1] );              /* b = 2*<p,q> */
26    double c = q[0]*q[0]+q[1]*q[1];                    /* c = |q|^2   */
27    double D = 4*a*c-b*b;
28    double root_D = 0.0;
29    double c_minus_a = c-a;
30    double G0 = 0.0;
31    double G1 = 0.0;
32
33    double Gn = 0.0;
34
35    assert(D>=-EPS*4*a*c);
36    if(D > EPS*4*a*c)
37      root_D = sqrt(D);
38    else
39      D=0.0;
40
41    if(D==0.0){
42      G0 = 2./c_minus_a;
43    }
44    else {
45      if(fabs(c_minus_a)<EPS*fabs(c))
46        G0 = M_PI/root_D;
47
48      else if(a<c)
49        G0 = 2.*atan(root_D/c_minus_a)/root_D;
50
51      else
52        G0 = 2.*(atan(root_D/c_minus_a)+M_PI)/root_D;
53    }
54    if(k>=1){
55      G1 = -b*G0;
56      if(a+b+c>EPS*a)
57        G1 += log(a+b+c);
58
59      if(a-b+c>EPS*a)
60        G1 -= log(a-b+c);
61
62      G1 /= (2.*a);
63
64      if(k==2)
65        return (2.-b*G1-c*G0)/a;
66      else if(k>2) {
67        Gn = -b*dlp(k-1,p,q) - c*dlp(k-2,p,q);
68        if(k % 2 == 0)
69          return (2.0/(1.0+k)+Gn)/a;
70        else /* k ungerade */
71          return Gn/a;
```

```
72      }
73      return G1;
74    }
75
76    return G0;
77  }
```

4.2.14 doubleDlp

Die Funktion berechnet

$$\int_{-1}^{1}\int_{-1}^{1} s^k t^\ell \frac{(\boldsymbol{w}+s\boldsymbol{u})\cdot n_{\boldsymbol{v}}}{|\boldsymbol{w}+s\boldsymbol{u}+t\boldsymbol{v}|^2}\,dt\,ds \quad \text{für } k,\ell\in\mathbb{N}_0 \text{ und } \boldsymbol{u},\boldsymbol{v},\boldsymbol{w}\in\mathbb{R}^2.$$

Für Details möchten wir auch hier auf [10, Lemma 3.1] verweisen.

Listing 28: doubleDlp

```
376  double doubleDlp(int k, int l, double* u, double* v, double* w, int A_eq_D, int
377  B_eq_C) {
378    double val = 0.0;
379
380    double lambda=0.0, mu=0.0, det=0.0;
381
382    double n[2]={0.0,0.0};
383    double w_plus_u[2]={0.0,0.0}, w_minus_u[2]={0.0,0.0};
384    double w_plus_v[2]={0.0,0.0}, w_minus_v[2]={0.0,0.0};
385
386    double dot_u_n=0.0, dot_w_n=0.0;
387    double dot_w_plus_u_n=0.0, dot_w_minus_u_n=0.0;
388
389    double hi = u[0]*u[0] + u[1]*u[1];
390    double hj = v[0]*v[0] + v[1]*v[1];
391
392    int mk = (k % 2 == 0) ? 1 : -1; /* if (-1)^k = 1 if k is even */
393    int ml = (l % 2 == 0) ? 1 : -1;
394
395    n[0] = v[1]/sqrt(hj);
396    n[1] = -v[0]/sqrt(hj);
397
398    w_plus_u[0] = w[0] + u[0];              w_plus_u[1] = w[1] + u[1];
399    w_minus_u[0] = w[0] - u[0];    w_minus_u[1] = w[1] - u[1];
400    w_plus_v[0] = w[0] + v[0];              w_plus_v[1] = w[1] + v[1];
401    w_minus_v[0] = w[0] - v[0];    w_minus_v[1] = w[1] - v[1];
402
403    dot_u_n = u[0]*n[0] + u[1]*n[1];
404    dot_w_n = w[0]*n[0] + w[1]*n[1];
405    dot_w_plus_u_n = w_plus_u[0]*n[0] + w_plus_u[1]*n[1];
406    dot_w_minus_u_n = w_minus_u[0]*n[0] + w_minus_u[1]*n[1];
407
408    det = u[0]*v[1] - u[1]*v[0];
409
410    /* u,v linearly dependent */
411    if(fabs(det) <= EPS*sqrt(hi*hj)) {
412      if(fabs(u[0])>fabs(u[1]))
413        mu = v[0]/u[0];
414      else
415        mu = v[1]/u[1];
416      if(fabs(dot_w_n) < EPS) { /* Hier muesste man auch checken koennen ob mu==1?*/
417        val = 0.0;
418    }
419    else
420      val = dot_w_n*( dlp(k,u,w_plus_v) + ml*dlp(k,u,w_minus_v) +
421      mu*(mk*dlp(l+1,v,w_minus_u)-dlp(l+1,v,w_plus_u)) );
422
423    if(k==0)
424      return val/( 1.0 + l );
425    else
426      return (val+k*mu*doubleDlp(k-1,l+1,u,v,w,A_eq_D,B_eq_C))/(1.0 + l);
427  }
```

```
428 else {
429   if(A_eq_D) {
430     val = 2*( dot_w_plus_u_n*dlp(l,v,w_plus_u) +
431     ml*(dot_u_n*dlp(k+1,u,w_minus_v) + dot_w_n*dlp(k,u,w_minus_v)));
432     if(k==0 && l==0)
433       return val;
434     else if(k==0)
435       return (val+l*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
436     else if(l==0)
437       return (val-k*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k+1);
438     else {
439       val = val - k*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C)
440       + l*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C);
441       return val/(1.0+k+l);
442     }
443   }
444   else if(B_eq_C) {
445     val = 2*( mk*dot_w_minus_u_n*dlp(l,v,w_minus_u) +
446     dot_u_n*dlp(k+1,u,w_plus_v)+dot_w_n*dlp(k,u,w_plus_v));
447     if(k==0 && l==0)
448       return val;
449     else if(k==0)
450       return (val-l*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
451     else if(l==0)
452       return (val+k*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k);
453     else {
454       val = val + k*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C)
455       - l*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C);
456       return val/(1.0+k+l);
457     }
458   }
459   else {
460     mu = (w[0]*v[1]-w[1]*v[0])/det;
461     /*mu = dot_w_n/dot_u_n;*/
462     lambda = (u[0]*w[1]-u[1]*w[0])/det;
463     /*lambda = (u[0]*w[1]-u[1]*w[0])/(u[0]*v[1]-u[1]*v[0]);*/
464 
465     val = (mu+1.0)*dot_w_plus_u_n*dlp(l,v,w_plus_u)
466     - mk*(mu-1.0)*dot_w_minus_u_n*dlp(l,v,w_minus_u) + (lambda+1.0) *
467     ( dot_u_n*dlp(k+1,u,w_plus_v) + dot_w_n*dlp(k,u,w_plus_v))
468     -ml*(lambda-1)*(dot_u_n*dlp(k+1,u,w_minus_v)
469     +dot_w_n*dlp(k,u,w_minus_v));
470 
471     if(k==0 && l==0)
472       return val;
473     else if(k==0)
474       return (val-l*lambda*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
475     else if(l==0)
476       return (val-k*mu*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k);
477     else {
478       val = val - k*mu*doubleDlp(k-1,l,u,v,w,A_eq_D,B_eq_C)
479       - l*lambda*doubleDlp(k,l-1,u,v,w,A_eq_D,B_eq_C);
480       return val/(1.0+k+l);
481     }
482   }
483 }
484 }
```

4.2.15 doubleDlpSwapped

Im Vergleich zu `doubleDlp` berechnet die Funktion

$$\int_{-1}^{1}\int_{-1}^{1} s^k t^\ell \frac{(\boldsymbol{w}+t\boldsymbol{v})\cdot n_{\boldsymbol{u}}}{|\boldsymbol{w}+s\boldsymbol{u}+t\boldsymbol{v}|^2}\,dt\,ds \quad \text{für } k,\ell\in\mathbb{N}_0 \text{ und } \boldsymbol{u},\boldsymbol{v},\boldsymbol{w}\in\mathbb{R}^2.$$

Dabei wurden die Rollen der Vektoren $\boldsymbol{u}$ und $\boldsymbol{v}$ vertauscht. Die tatsächliche Berechnung läuft dann wieder ähnlich wie bei der Funktion `doubleDlp` ab.

Listing 29: doubleDlpSwapped

```
486 double doubleDlpSwapped(int k, int l, double* u, double* v, double* w,
487         int A_eq_D, int B_eq_C) {
488   double val = 0.0;
489
490   double lambda=0.0, mu=0.0, det=0.0;
491
492   double n[2]={0.0,0.0};
493   double w_plus_u[2]={0.0,0.0}, w_minus_u[2]={0.0,0.0};
494   double w_plus_v[2]={0.0,0.0}, w_minus_v[2]={0.0,0.0};
495
496   double dot_v_n=0.0, dot_w_n=0.0;
497   double dot_w_plus_v_n=0.0, dot_w_minus_v_n=0.0;
498
499   double hi = u[0]*u[0] + u[1]*u[1];
500   double hj = v[0]*v[0] + v[1]*v[1];
501
502   int mk = (k % 2 == 0) ? 1 : -1; /* if (-1)^k = 1 if k is even */
503   int ml = (l % 2 == 0) ? 1 : -1;
504
505   n[0] = u[1]/sqrt(hi); n[1] = -u[0]/sqrt(hi);
506
507   w_plus_u[0] = w[0] + u[0];           w_plus_u[1] = w[1] + u[1];
508   w_minus_u[0] = w[0] - u[0];   w_minus_u[1] = w[1] - u[1];
509   w_plus_v[0] = w[0] + v[0];           w_plus_v[1] = w[1] + v[1];
510   w_minus_v[0] = w[0] - v[0];   w_minus_v[1] = w[1] - v[1];
511
512   dot_v_n = v[0]*n[0] + v[1]*n[1];
513   dot_w_n = w[0]*n[0] + w[1]*n[1];
514   dot_w_plus_v_n = w_plus_v[0]*n[0] + w_plus_v[1]*n[1];
515   dot_w_minus_v_n = w_minus_v[0]*n[0] + w_minus_v[1]*n[1];
516
517   det = u[0]*v[1] - u[1]*v[0];
518
519   /* u,v linearly dependent */
520 if(fabs(det) <= EPS*sqrt(hi*hj)) {
521     if(fabs(u[0])>fabs(u[1]))
522       mu = v[0]/u[0];
523     else
524       mu = v[1]/u[1];
525     if(fabs(dot_w_n) < EPS) { /* Hier muesste man auch checken koennen ob mu==1?*/
526       val = 0.0;
527     }
528     else
529       val = dot_w_n*( dlp(k,u,w_plus_v) + ml*dlp(k,u,w_minus_v) +
530       mu*(mk*dlp(l+1,v,w_minus_u)-dlp(l+1,v,w_plus_u)) );
531
532   if(k==0)
533     return val/( 1.0 + l );
534   else
535     return (val+k*mu*doubleDlpSwapped(k-1,l+1,u,v,w,A_eq_D,B_eq_C))/(1.0 + l);
536 }
537 else {
538   if(A_eq_D) {
539     val = 2*(mk*(dot_v_n*dlp(l+1,v,w_minus_u) + dot_w_n*dlp(l,v,w_minus_u))
540           + dot_w_plus_v_n*dlp(k,u,w_plus_v) );
541     if(k==0 && l==0)
542       return val;
543     else if(k==0)
544       return (val-l*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
545     else if(l==0)
546       return (val+k*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k);
547     else {
548       val = val + k*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C)
549       - l*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C);
550       return val/(1.0+k+l);
551     }
552   }
553   else if(B_eq_C) {
554     val = 2*(dot_v_n*dlp(l+1,v,w_plus_u) + dot_w_n*dlp(l,v,w_plus_u) +
555     ml*dot_w_minus_v_n*dlp(k,u,w_minus_v) );
556     if(k==0 && l==0)
557       return val;
558     else if(k==0)
559       return (val+l*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
```

```
560       else if(l==0)
561         return (val-k*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k);
562       else {
563         val = val - k*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C)
564         + l*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C);
565         return val/(1.0+k+l);
566       }
567     }
568     else {
569       mu = (w[0]*v[1]-w[1]*v[0])/det;
570       /*mu = dot_w_n/dot_u_n;*/
571       lambda = (u[0]*w[1]-u[1]*w[0])/det;
572       /*lambda = (u[0]*w[1]-u[1]*w[0])/(u[0]*v[1]-u[1]*v[0]);*/
573
574       val = (mu+1.0)*(dot_v_n*dlp(l+1,v,w_plus_u) + dot_w_n*dlp(l,v,w_plus_u) )
575           - mk*(mu-1.0)*(dot_v_n*dlp(l+1,v,w_minus_u) + dot_w_n*dlp(l,v,w_minus_u) )
576           + (lambda+1.0)*dot_w_plus_v_n*dlp(k,u,w_plus_v)
577           - ml*(lambda-1.0)*dot_w_minus_v_n*dlp(k,u,w_minus_v);
578
579       if(k==0 && l==0)
580         return val;
581       else if(k==0)
582         return (val-l*lambda*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C))/(1.0+l);
583       else if(l==0)
584         return (val-k*mu*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C))/(1.0+k);
585       else {
586         val = val - k*mu*doubleDlpSwapped(k-1,l,u,v,w,A_eq_D,B_eq_C)
587         - l*lambda*doubleDlpSwapped(k,l-1,u,v,w,A_eq_D,B_eq_C);
588         return val/(1.0+k+l);
589       }
590     }
591   }
592   }
```

Literatur

[1] Mayr Markus: *Stabile Implementierung der Randelementmethode auf stark adaptierten Netzen,* Bachelorarbeit, 2010.

[2] Börm Steffen und Grasedyck Lars: *Low-Rank Approximation of Integral Operators by Interpolation,* Computing 72, 325-32, Springer-Verlag, 2004.

[3] Börm Steffen, Löhndorf Maike und Melenk Jens Markus: *Approximation of integral operators by variable-order interpolation,* Numer. Math., 99(4):605-643, 2005.

[4] Plato Robert: *Numerische Mathematik kompakt,* Vieweg+Teubner Verlag, 2010.

[5] Kaltenbäck Michael: *Analysis 3 für Technische Mathematik, WS 2012/2013,* Vorlesungsskriptum, Technische Universität Wien, 2012.

[6] Auzinger Winfried, Koch Othmar und Praetorius Dirk: *Numerische Mathematik,* Vorlesungsskriptum, Technische Universität Wien, 2012.

[7] Rivlin J. Theodore: *The Chebyshev Polynomials,* John Wiley & Sons, 1974.

[8] Königsberger Konrad: *Analysis 2,* Springer-Verlag, 2004

[9] Praetorius Dirk: *Introduction to Boundary Element Method,* Technische Universität Wien, 2007.

[10] Maischak Matthias: *The analytical computation of the Galerkin elements for the Laplace, Lamé and Helmholtz equation in 2D-BEM,* Institut für Angewandte Mathematik, Leibnitz Universität Hannover, 2001.

[11] Aurada Markus, Ebner Michael, Feischl Michael, Ferraz-Leite Samuel, Führer Thomas, Goldenits Petra, Karkulik Michael, Mayr Markus, und Praetorius Dirk: *A* Matlab *implementiation of adaptive 2D-BEM,* ASC Report, 24/2011, Technische Universität Wien, 2011.

[12] Aurada Markus, Ebner Michael, Feischl Michael, Ferraz-Leite Samuel, Führer Thomas, Goldenits Petra, Karkulik Michael, Mayr Markus, und Praetorius Dirk: *Hilbert (Release 3): A Matlab implementiation of adaptive BEM,* `http://www.asc.tuwien.ac.at/abem/hilbert/`, 2013.

[13] Henry O. Jacobs: *How to stare at the higher-order n-dimensional chain rule without losing your marbles,* arXiv:1410.3493v3 [math.CA]

[14] Carstensen Carsten, Faermann Birgit: *Mathematical foundation of a posteriori error estimates and adaptive mesh-refining algorithms for boundary integral equations of the first kind,* Elsevier Science Ltd., 2001.

[15] Hackbusch Wolfgang *Hierarchische Matrizen,* Springer-Verlag Berlin Heidelberg, 2009.