BACHELORARBEIT

Adaptive isogeometrische Finite-Elemente-Methode mit T-Splines

ausgeführt am

Institut für
Analysis und Scientific Computing
TU Wien

unter der Anleitung von

Univ.Prof. Dipl.-Math. Dr.techn. Dirk Praetorius
Dr.techn. Gregor Gantner

durch

Felix Blödorn

Wien, am 24. November 2020

Danksagung

In diesem ersten Satz vorliegender Arbeit möchte ich mich ganz herzlich bei *Univ.Prof. Dipl.-Math. Dr.techn. Dirk Praetorius* für die Idee und Betreuung dieser Arbeit bedanken. Seine unzähligen sorgfältigen Korrekturen und Verbesserungsvorschläge haben ihr zu Struktur und Form verholfen und waren für mich äußerst lehrreich. Weiter möchte ich *Dr.techn. Gregor Gantner* für seine wertvollen Erklärungen und seine unermüdliche Hilfe, speziell bei Überprüfung und Fehlersuche des Codes danken. Die stets positive und respektvolle Zusammenarbeit habe ich sehr geschätzt.

Ich möchte meiner gesamten Familie danken, allen voran meinen liebevollen Eltern Karin und Heinz für ihre Unterstützung, auf die ich stets zählen kann, ihre Geduld und dafür, dass sie immer für mich da sind.

Dana, du bist meine große Motivation und gibst mir Energie und Lebensfreude.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 24. November 2020

Felix Blödorn

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit gibt zuerst eine kompakte theoretische Einführung zu T-Splines und adaptiver IGAFEM, um sich dann der Implementierung dieser zu widmen. Zu Beginn werden in Kapitel 2 multivariate B-Splines erklärt. Das sind stückweise definierte Polynome auf Tensor-Gittern. Visuell gesprochen besteht ein solches Gitter aus Rechtecken, die ausschließlich Seite an Seite liegen und nicht Ecke an Seite, siehe hierfür Abbildung 2.3. Bei den in Kapitel 3 eingeführten, darauf basierenden T-Splines wird diese Beschränkung gelockert und es sind auch gewisse Eckpunkte mit Valenz drei erlaubt, siehe Abbildung 3.4. Sie liefern flexible Gitter, die sogenannten T-Gitter und damit deutlich effizientere Resultate, da sie lokale Netzverfeinerung ermöglichen. Diese Einführung folgt der Notation und dem Inhalt von [2].

Nicht auf allen T-Gittern sind die assoziierten T-Spline-Kumulationsfunktionen linear unabhängig, siehe [5] für ein Gegenbeispiel. Deswegen wurden im selben Artikel die AST-Splines[1] eingeführt. Die AST-Splines stellen eine Teilmenge der T-Splines dar und sind zentraler Bestandteil dieser Arbeit. Morgenstern [9] hat dazu 2017 einen effizienten Algorithmus vorgestellt, der AST-Gitter mit beliebigen markierten Elementen in feinere T-Gitter überführt, die ebenfalls AS[2] sind. In Kapitel 4 wird eine Netz-adaptive FEM[3] kombiniert mit Methoden der IGA[4] präsentiert, nämlich IGAFEM, und deren Anwendung auf das Poisson-Problem erklärt.

Die letzten zwei Kapitel widmen sich dem eigentlichen Herzstück der Arbeit, nämlich einer effizienten Implementierung der adaptiven IGAFEM mit T-Splines in MATLAB und der Präsentation einiger damit berechneter Beispiele. Dabei werden die wichtigen Erfolgs-Parameter vorgestellt: Fehlerschätzer η, Energiefehler δE und maximaler Punktfehler, um die Konvergenz zur richtigen Lösung zu überprüfen.

[1] Analysis-suitable T-Splines (AST-Splines)
[2] Analysis-suitable, AS
[3] Finite-Elemente-Methode, FEM
[4] Isogeometrische Analysis, IGA

2. B-Splines

B-Splines[1] sind spezielle Polynome vom Grad p, die rekursiv nach der Formel von Cox[2] und de Boor[3] definiert werden, vgl. [11]. Nach kurzer Erklärung und Visualisierung einer freien Kurve im dreidimensionalen Raum wird auf den multivariaten Fall erweitert. Damit können bereits beliebige Ober- und Hyperflächen beliebig genau parametrisiert werden.

2.1. Univariate B-Splines

Angefangen mit dem p-offenen Knotenvektor werden in diesem Abschnitt die elementaren Definitionen für B-Splines gegeben. Es folgt die zentrale Definition der B-Splines, nämlich die Cox-de-Boor-Formel, sowie einige wichtige Eigenschaften von B-Splines. Die erste praktische Anwendung (und Visualisierung) von univariaten B-Splines wird in der Parametrisierung von freien, glatten Raumkurven gezeigt. Am Schluss des Abschnitts werden Algorithmen zu verschiedenen Möglichkeiten der Verfeinerung vorgestellt. Diese Einführung ist aus [2] entnommen.

2.1.1. Definitionen

Definition 2.1 (Offener Knotenvektor). *Mit $n, p \in \mathbb{N}_0$ wird ein gegebener, geordneter Knotenvektor*

$$\Xi := (\xi_1, \xi_2, ..., \xi_{n+p+1})$$

p-offen genannt, wenn gilt

$$\xi_1 = \xi_2 = ... = \xi_{p+1} < \xi_{p+2} \leq \xi_{p+3} \leq ... \leq \xi_{n-1} < \xi_n = \xi_{n+1} = ... = \xi_{n+p+1}.$$

In Definition 2.1 haben Start- und Endknoten Multiplizität $p + 1$. Für bessere Handhabung und ohne Beschränkung der Allgemeinheit wählen wir $\xi_1 = 0$ und $\xi_{n+p+1} = 1$, was einfach auf allgemeine Intervallen $[a, b]$ abgebildet werden kann mithilfe der affinen Transformation

$$\varphi : [0, 1] \to [a, b], \quad \varphi(t) = a + t(b - a).$$

[1] Basis-Splines (B-Splines)
[2] Maurice G. Cox, geb. 1940 in Epping (Sussex), Großbritannien
[3] Carl-Wilhelm Reinhold de Boor, geb. 1937 in Stolp, damals Deutschland

Der Knotenvektor ohne Wiederholungen wird durch $Z = \{\zeta_1, \zeta_2, ..., \zeta_N\}$, sowie die Multiplizität jedes Knotens ζ_j durch m_j notiert, sodass

$$\Xi = (\underbrace{\zeta_1, \zeta_1, ..., \zeta_1}_{m_1\text{-mal}}, \underbrace{\zeta_2, \zeta_2, ..., \zeta_2}_{m_2\text{-mal}}, ..., \underbrace{\zeta_N, \zeta_N, ..., \zeta_N}_{m_N\text{-mal}})$$

gilt. Das (eindimensionale) Gitter $\mathcal{G}$ wird definiert durch

$$\mathcal{G} := \{[\zeta_1, \zeta_2], [\zeta_2, \zeta_3], ...[\zeta_{N-1}, \zeta_N]\},$$

was erst später wichtig werden wird, sobald wir auf höhere Dimensionen erweitern. Basierend auf dem Knotenvektor Ξ werden B-Splines nach der folgenden rekursiven Formel von Cox und de Boor definiert:

Definition 2.2 (Cox-de-Boor-Formel). *Für Polynomgrad $p \in \mathbb{N}$ und p-offenen Knotenvektor Ξ laut Definition 2.1 sind B-Splines $B_{i,p}$ definiert durch*

$$B_{i,0}(\zeta) = \begin{cases} 1 & \text{wenn } \xi_i \leq \zeta < \xi_{i+1}, \\ 0 & \text{sonst} \end{cases}$$

für Grad $p = 0$ und

$$B_{i,p}(\zeta) = \frac{\zeta - \xi_i}{\xi_{i+p} - \xi_i} B_{i,p-1}(\zeta) + \frac{\xi_{i+p+1} - \zeta}{\xi_{i+p+1} - \xi_{i+1}} B_{i+1,p-1}(\zeta)$$

für höheren Grad $p \geq 1$. Formal wird $(\cdot)/0$ hier als 0 definiert.

Plots von B-Splines vom Grad $p = 0, 1, 2, 3$ sind gegeben in Abbildung 2.1. Alle B-Splines $B_{i,p}$ nach Definition 2.2 sind

- eine Zerlegung der Eins ($\sum_i B_{i,p}(\zeta) \equiv 1$) nach Konstruktion,
- nichtnegativ ($B_{i,p} \geq 0$) auf dem gesamten Definitionsbereich und
- an den Knotenpunkten ξ_i mit Multiplizität m_i genau $(p - m_i)$-fach stetig differenzierbar.

Der Fall $m_i = p+1$, also (-1)-Differenzierbarkeit, steht für Unstetigkeit am Knotenpunkt ξ_i. Es wird angenommen, dass $m_i \leq p + 1$ gilt, denn $m_i = p + 2$ würde eine Funktion bereitstellen, die identisch Null ist.

Definition 2.3 (Univariater B-Spline-Raum). *Für gegebenen Polynomgrad p und KnotenvektorΞ wird der Raum der univariaten B-Splines durch ebendiese aufgespannt:*

$$S_p(\Xi) = span\{B_{i,p}, i = 1, 2, ..., n\}.$$

Jede B-Spline-Funktion $B_{i,p}$ hängt nur von $p+2$ Knoten ab, welche im lokalen Knotenvektor zusammengefasst sind.

Definition 2.4 (Lokaler Knotenvektor). *Für jede B-Spline-Funktion $B_{i,p}$ nach Definition 2.2 wird der lokale Knotenvektor durch*

$$\Xi_{i,p} := \{\xi_i, \xi_{i+1}, ..., \xi_{i+p+1}\}$$

definiert.

Der Träger jeder B-Spline-Funktion ist exakt

$$\text{supp}(B_{i,p}) = [\xi_i, \xi_{i+p+1}],$$

was in der alternativen Notation $B[\Xi_{i,p}](\zeta) := B_{i,p}(\zeta)$ noch deutlicher wird. Bei gegebener Knotenspannweite $I_j := (\zeta_j, \zeta_{j+1})$, welche mit genau einem speziellen Intervall (ξ_i, ξ_{i+1}) übereinstimmt, wird die Trägererweiterung

$$\widetilde{I_j} := (\xi_{i-p}, \xi_{i+p+1})$$

definiert. Sie ist genau das Innere der Vereinigung der Träger jener B-Spline-Funktionen, deren Träger I_j schneidet.

2.1.2. Freie Raumkurve

Eine B-Spline-Kurve im $\mathbb{R}^m$ mit gegebenen Kontrollpunkten $\mathbf{c}_i \in \mathbb{R}^m$ ist definiert durch

$$\mathbf{S}(\zeta) = \sum_{i=1}^{n} \mathbf{c}_i B_{i,p}(\zeta).$$

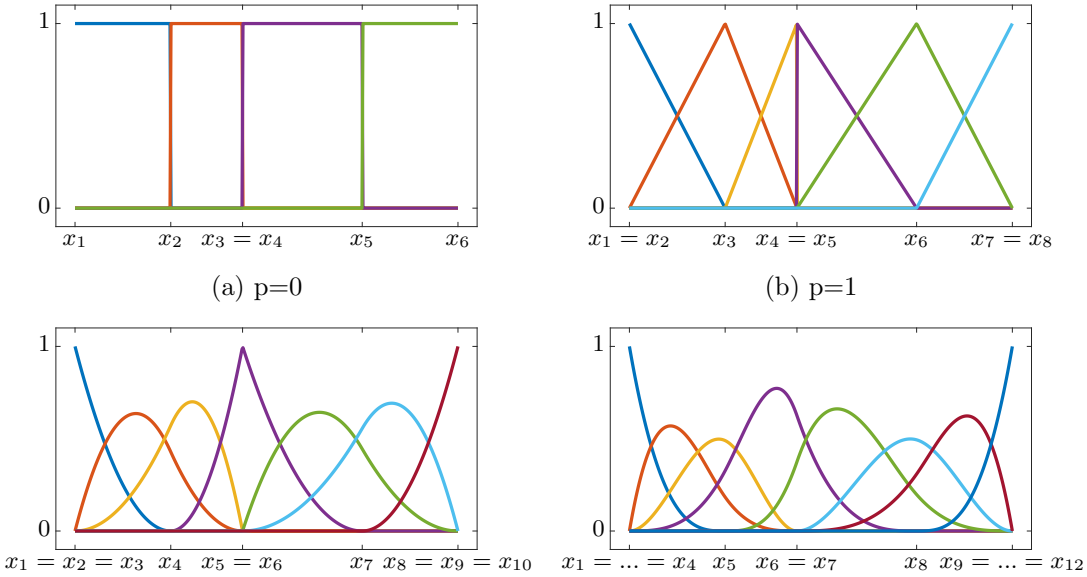
Eine B-Spline-Kurve mit $p = 3$ und einfacher Multiplizität aller inneren Punkte im $\mathbb{R}^2$ mit dem dazugehörigen Kontrollpolygon, also der stückweise linearen Verbindung der Kontrollpunkte $\mathbf{c}_i$, ist zu sehen in Abbildung 2.2.

2.1.3. Einfügen von Knoten und Erhöhung des Polynomgrads

Im Kontext von geometrischer Modellierung bedeutet Verfeinerung die Möglichkeit, neue Knoten in eine B-Spline-Kurve einfügen zu können, ohne dabei die Parametrisierung $\mathbf{S}(\zeta)$ ändern zu müssen. Dafür muss ein verfeinerter Raum definiert werden, der den Raum der Parametrisierung enthält. Mit gegebenem B-Spline-Raum $S_{p^0}(\Xi^0)$ nennen wir $S_p(\Xi)$ eine Verfeinerung von $S_{p^0}(\Xi^0)$, wenn

$$S_{p^0}(\Xi^0) \subset S_p(\Xi).$$

Für B-Splines wird eine Verfeinerung erreicht mittels Einfügen von Knoten, Erhöhung des Polynomgrads oder der Kombination beider.



(a) p=0 (b) p=1

(c) p=2 (d) p=3

Abbildung 2.1.: Splines für verschiedene Polynomgrade p für den jeweils p-offenen Knotenvektor Ξ mit einem doppelten inneren Knotenpunkt. An diesem ist die Funktion nur $\mathcal{C}^{p-2}$ anstelle von $\mathcal{C}^{p-1}$, wie an allen anderen Stellen.

(i). *h-Verfeinerung* (analog zu Gitterverfeinerung) wird durch Einfügen von Knoten erreicht. Sei $\bar{\Xi} := \Xi \cup \{\bar{\xi}\} = (\xi_1, \xi_2, \ldots, \xi_j, \bar{\xi}, \xi_{j+1}, \ldots, \xi_{n+p+1})$ der Knotenvektor nach Einfügen von $\bar{\xi}$ in $\Xi = (\xi_1, \xi_2, \ldots, \xi_{n+p+1})$ sowie $\xi_j \leq \bar{\xi} \leq \xi_{j+1}$. Bezeichnet man die Knotenpunkte in $\bar{\Xi}$ mit $\bar{\xi}_k$ und die dazugehörigen B-Spline-Funktionen mit $\bar{B}_{i,p}$, kann man die alten B-Spline-Funktionen mittels der neuen ausdrücken:

$$B_{i,p}(\zeta) = \alpha_i \bar{B}_{i,p}(\zeta) + (1 - \alpha_{i+1}) \bar{B}_{i+1,p}(\zeta)$$

mit den Koeffizienten (siehe [2])

$$\alpha_i = \begin{cases} 1 & \text{für } i = 1, 2, ..., j-p, \\ \frac{\bar{\xi} - \bar{\xi}_i}{\bar{\xi}_{i+p+1} - \bar{\xi}_i} & \text{für } i = j-p+1, j-p+2, ..., j, \\ 0 & \text{für } i = j+1, j+2, ..., n+1. \end{cases}$$

(ii). *p-Verfeinerung* entspricht der Erhöhung des Polynomgrads mit festen Knotenpunkten.

(iii). *k-Verfeinerung* entspricht Erhöhung des Polynomgrads und anschließendem Einfügen eines Knotenpunktes. Dieser Algorithmus sichert die Regularität aller Knotenpunkte.

Ein Algorithmus zur h-Verfeinerung, der mehr als einen Knoten pro Durchlauf einfügen kann, ist in [7] zu finden.

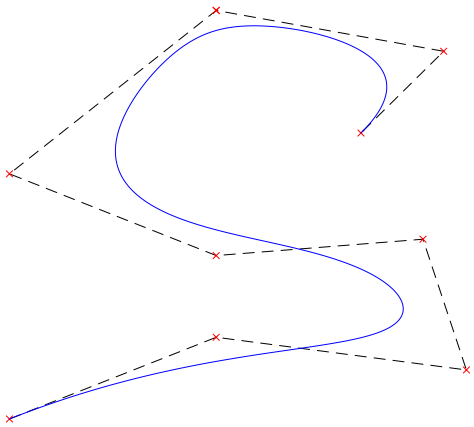


Abbildung 2.2.: Kubische B-Spline-Kurve (gezogen), Stützpunkte $\mathbf{c}_i$ (Kreuze) und Kontrollpolygon (strichliert)

2.2. Multivariate B-Splines

Multivariate B-Splines werden in natürlicher Weise über das Tensorprodukt von univariaten B-Splines definiert. Die restlichen Definitionen werden ebenfalls durch entsprechendes Produkt auf beliebige Dimension erweitert. Ein Beispiel, wie bereits mit geringem Polynomgraden und wenigen Gitterpunkten gegebene Oberflächen gut angenähert werden können, wird am Schluss des Abschnitts gegeben. Dieser Teil ist entnommen aus [2].

2.2.1. Definitionen

Definition 2.5 (Knotentensor). *Mit Raumdimension d seien die Vektorlänge $n_\ell \in \mathbb{N}$, der Polynomgrad $p_\ell \in \mathbb{N}$ sowie der p_ℓ-offene Knotenvektor $\Xi_\ell = (\xi_{\ell,1}, \xi_{\ell,2}, ..., \xi_{\ell,n_\ell+p_\ell+1})$ gegeben für $\ell = 1, 2, ..., d$. Die Polynomgrade werden im Polynomgradvektor*

$$\mathbf{p} = (p_1, p_2, ..., p_d)$$

zusammengefasst. Aus dem Tensorprodukt der Ξ_ℓ ergibt sich der Knotentensor

$$\boldsymbol{\Xi} = \Xi_1 \times \Xi_2 \times ... \times \Xi_d.$$

Definition 2.6 (Bézier-Gitter). *Über die $Z_\ell = \{\zeta_{\ell,1}, \zeta_{\ell,2}, ..., \zeta_{\ell,N_\ell}\}$, die jeweils aus Ξ_ℓ konstruiert werden, wird in der Parameter-Basis $\Omega = (0,1)^d$) das karthesisches Gitter*

$$\mathcal{M} = \{Q_{\mathbf{j}} = I_{1,j_1} \times I_{2,j_2} \times ... \times I_{d,j_d} : I_{\ell,j_\ell} = (\zeta_{\ell,j_\ell}, \zeta_{\ell,j_\ell+1}) \text{ für } 1 \leq j_\ell \leq N_\ell - 1\}$$

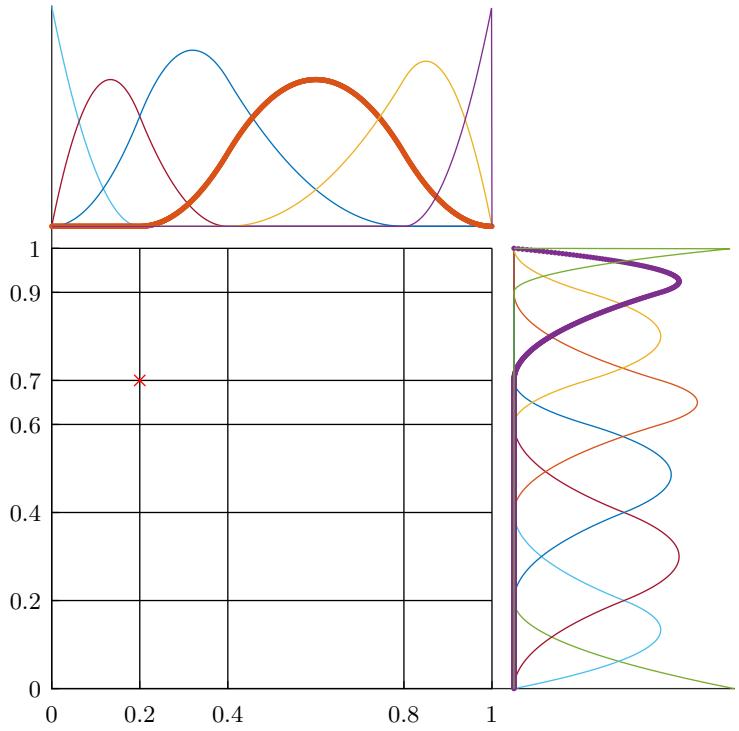


Abbildung 2.3.: Zweidimensionales Bézier-Gitter in Parameter-Basis $\Omega = (0,1)^2$ in der Mitte, oben und rechts jeweils die B-Spline-Funktionen vom Grad $p = q = 2$ in x- und y-Richtung. Zum Punkt $[x = 0{,}2; y = 0{,}7]$ gehören beispielsweise der rote Graph mit dem Maximum bei $x \approx 0{,}6$ in x-Richtung und der violette Graph mit dem Maximum bei $y \approx 0{,}9$ in y-Richtung

gebildet, welches als Bézier-Gitter[4] *bezeichnet wird.*

Ein beispielhaftes, zweidimensionales Bézier-Gitter ist in Abbildung 2.3 zu sehen. Für ein allgemeines Bézier-Element $Q_{\mathbf{j}} \in \mathcal{M}$ wird (analog zum univariaten Fall) ihre Trägererweiterung durch

$$\widetilde{Q}_{\mathbf{j}} = \widetilde{I}_{1,j_1} \times \widetilde{I}_{2,j_2} \times ... \times \widetilde{I}_{d,j_d}$$

festgelegt. Multivariate B-Splines werden über das Produkt von bestimmten univariaten B-Splines definiert.

Definition 2.7 (Lokaler Knotentensor). *Aus der Menge der Multi-Indizes* $\mathbf{I} = \{\mathbf{i} = (i_1, i_2, ..., i_d) : 1 \leq i_\ell \leq n_\ell\}$ *wird für jeden Multi-Index* $\mathbf{i} = (i_1, i_2, ..., i_d)$ *der dazugehörige*

[4] Pierre Étienne Bézier, geboren 1910 in Paris, gestorben 1999 ebenda

lokale Knotentensor durch

$$\mathbf{\Xi_{i,p}} = \Xi_{i_1,p_1} \times \Xi_{i_2,p_2} \times ... \times \Xi_{i_d,p_d}$$

definiert.

Mithilfe der bisherigen Definitionen können nun multivariate B-Splines definiert werden.

Definition 2.8 (Multivariate B-Splines). *Mit gegebenen Knotentensor* $\mathbf{\Xi}$ *und Multi-Index* $\mathbf{i} = (i_1, i_2, \ldots, i_d)$ *aus* $\mathbf{I}$ *wird die B-Spline-Funktion durch*

$$B_{\mathbf{i},\mathbf{p}}(\zeta) = B[\Xi_{i_1,p_1}](\zeta_1) \cdot B[\Xi_{i_2,p_2}](\zeta_2) \cdots B[\Xi_{i_d,p_d}](\zeta_d)$$

definiert.

Wie B-Spline-Funktionen mehreren Punkten zugeordnet werden, abhängig von deren Koordinaten, ist exemplarisch in Abbildung 2.3 gezeigt.

Definition 2.9 (B-Spline-Raum). *Für die Menge der multivariaten B-Splines nach Definition 2.8 ist der B-Spline-Raum in der Parameter-Basis* Ω *gegeben durch*

$$S_{\mathbf{p}}(\mathbf{\Xi}) = \ span\{B_{\mathbf{i},\mathbf{p}}(\zeta) : \mathbf{i} \in \mathbf{I}\}.$$

Der B-Spline-Raum aus Definition 2.9 ist der Raum der stückweise definierten Polynome vom Grad p mit der Bézier-Element-Regularität gegeben durch die Knoten-Multiplizität.

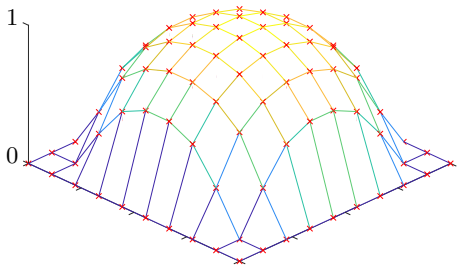
2.2.2. Freie Raumgeometrien

Analog zur Raumkurve im univariaten Fall können multivariate Geometrien im $\mathbb{R}^m$ definiert werden. Eine B-Spline-Parameterisierung ist eine beliebige Linearkombination der B-Spline-Basisfunktionen mit Kontrollpunkten $\mathbf{c_i} \in \mathbb{R}^n$, also

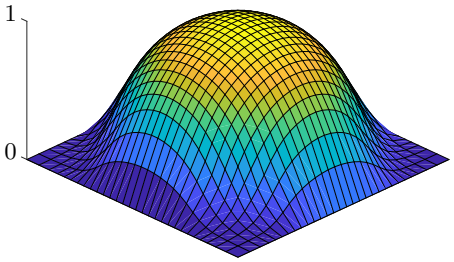
$$\mathbf{S}(\zeta) = \sum_{\mathbf{i} \in \mathbf{I}} \mathbf{c_i} B_{\mathbf{i},\mathbf{p}}(\zeta), \quad \text{mit } \zeta \in \mathbf{\Omega} := [0,1]^d.$$

Abhängig von d und m kann das eine ebene Oberfläche im $\mathbb{R}^2$ ($d = 2, m = 2$), eine beliebig gekrümmte Oberfläche im $\mathbb{R}^3$ ($d = 2, m = 3$) oder ein Volumen im $\mathbb{R}^3$ ($d = 3, m = 3$) sein. Der zweite Fall, nämlich eine Fläche im Raum, ist in Abbildung 2.4 zu sehen.

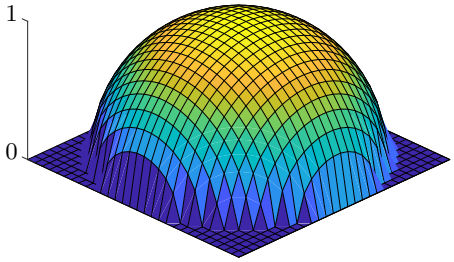
Verfeinerungsalgorithmen zum Einfügen von Knoten und Polynomgraderhöhung lassen sich auf multivariate B-Splines verallgemeinern. Wie auch im univariaten Fall ist bei gegebenem B-Spline-Raum $S_{\mathbf{p^0}}(\mathbf{\Xi^0})$ der Raum $S_{\mathbf{p}}(\mathbf{\Xi})$ eine Verfeinerung, genau dann wenn $S_{\mathbf{p^0}}(\mathbf{\Xi^0}) \subset S_{\mathbf{p}}(\mathbf{\Xi})$ gilt. Das Kontrollgitter (univariat: Kontrollpolygon) ist ebenfalls die stückweise lineare Verbindung aller benachbarten Kontrollpunkte, wie jenes in Abbildung 2.4a.



(a) Kontrollgitter mit Stützpunkten



(b) B-Spline-Oberfläche mit $p_1 = p_2 = 2$



(c) Originalbild

Abbildung 2.4.: Hemisphäre auf einer Ebene, Gitter mit 10×10 Knoten. Einfache Interpolation: Festlegung der Knotenpunkte durch Auswertung an den Anker-Koordinaten.

3. T-Splines und lokale Verfeinerung

Der namensgebende Unterschied zu B-Splines sind Ecken der Valenz drei, siehe Abbildung 3.4. Das ist das Aufeinandertreffen der Ecken zweier Rechtecke mit der Seite eines dritten Rechtecks, je nach Orientierung notiert durch $\top, \dashv, \perp$ oder $\vdash$. Das Bézier-Gitter wird in Abschnitt 3.1 erweitert und zweckgemäß in einen aktiven und einen Randbereich geteilt. Das Konzept der Anker wird in Abschnitt 3.2 eingeführt, schließlich sind im Allgemeinen nicht mehr alle Knotenpunkte eines vergleichbaren Bézier-Gitters mit einer B-Splinefunktion verknüpft.

Aufgrund mangelnder linearer Unahängigkeit der assoziierten T-Spline-Kumulationsfunktionen allgemeiner T-Spline-Gittern (vgl. [5]) wird in Abschnitt 3.3 die Teilmenge der AST-Splines eingeführt. Die Darstellung wird auf den kompakteren, zweidimensionalen Fall beschränkt, die Erweiterung auf höhere Dimensionen wird mit analogen Eigenschaften in [1] beschrieben.

Schließlich wird im letzten Abschnitt 3.4 ein ebenfalls auf höhere Dimensionen erweiterbarer Algorithmus von Morgenstern[1] und Peterseim[2] vorgestellt, der die AS-Eigenschaft erhält.

Abschnitte 3.1 - 3.3 sind aus [2] entnommen, die Notation folgt ebenfalls weitestgehend diesem Artikel.

3.1. T-Gitter

Definition 3.1 (T-Gitter). *Ein T-Gitter $\mathcal{T}$ in der Index-Basis ist eine Rechtecks-Zerlegung des Index-Definitionsbereichs*

$$[\underline{n_1}, \overline{n_1}] \times [\underline{n_2}, \overline{n_2}],$$

mit $\underline{n_1}, \overline{n_1}, \underline{n_2}, \overline{n_2} \in \mathbb{Z}$, sodass die Eckpunkte[3] aller Elemente[4] $T \in \mathcal{T}$ ganzzahlige Koordinaten haben. Präziser gesprochen ist $\mathcal{T}$ genau die Menge aller Elemente obiger Zerlegung.

Es werden (dem englischsprachigen Literatur-Standard folgend)

[1] Philipp Morgenstern
[2] Daniel Peterseim, geb. 1980 in Mühlhausen/Thüringen
[3] Eckpunkte, engl. Vertices $V \in \mathcal{V}$
[4] Elemente, engl. Tiles $T \in \mathcal{T}$

- mit $T \in \mathcal{T}$ die Elemente,
- mit $E \in \mathcal{E}$ die Kanten und
- mit $V \in \mathcal{V}$ die Eckpunkte

des T-Gitters bezeichnet. $\mathcal{E}$ bzw. $\mathcal{V}$ ist dabei die Menge aller Kanten[5] bzw. Eckpunkte. Eine Kante E ist die Verbindung zweier Eckpunkte, die kein Element aus $\mathcal{T}$ schneidet. Es wird angenommen, dass sie keine Eckpunkte aus $\mathcal{V}$ enthält und an ihren Endpunkten offen ist. Die Menge aller vertikalen bzw. horizontalen Kanten wird mit $\mathcal{E}^v$ bzw. $\mathcal{E}^h$ bezeichnet, die Vereinigung beider ist $\mathcal{E} := \mathcal{E}^v \cup \mathcal{E}^h$. Die Menge aller Ecken V und Kanten E eines Elements T, also sein Rand, wird mit ∂T bezeichnet.

Die Vereinigung beider Eckpunkte $V \in \mathcal{V}$ einer Kante E wird mit ∂E notiert. Die Valenz eines Eckpunktes $V \in \mathcal{V}$ ist die Anzahl der Kanten $E \in \mathcal{E}$, für die $V \subset \partial E$ gilt. Da alle Elemente $T \in \mathcal{T}$ Rechtecke sind, ist für alle inneren Eckpunkte $V \in]\underline{n_1}, \overline{n_1}[\times]\underline{n_2}, \overline{n_2}[$ entweder Valenz drei oder vier möglich. Das vertikale bzw. horizontale Skelett $\mathcal{S}^v$ bzw. $\mathcal{S}^h$ ist die Vereinigung aller Eckpunkte und vertikalen bzw. horizontalen Kanten. Die Vereinigung beider wird als Skelett[6] $\mathcal{S} := \mathcal{S}^v \cup \mathcal{S}^h$ bezeichnet.

Definition 3.2 (Gitterbereiche). *Mit gegebenen Polynomgraden $\mathbf{p} = (p_1, p_2)$ wird der Index-Definitionsbereich$[\underline{n_1}, \overline{n_1}] \times [\underline{n_2}, \overline{n_2}]$ in zwei Bereiche geteilt, nämlich einen aktiven Bereich[7] $AR_{\mathbf{p}}$ und in einen Randbereich[8] $FR_{\mathbf{p}}$, sodass*

$$\begin{aligned} AR_{\mathbf{p}} = [\underline{n_1} + \lfloor (p_1+1)/2 \rfloor, \overline{n_1} - \lfloor (p_1+1)/2 \rfloor] \times \\ \times [\underline{n_2} + \lfloor (p_2+1)/2 \rfloor, \overline{n_2} - \lfloor (p_2+1)/2 \rfloor] \end{aligned}$$

und

$$\begin{aligned} FR_{\mathbf{p}} = ([\underline{n_1}, \underline{n_1} + \lfloor (p_1+1)/2 \rfloor] \cup [\overline{n_1} - \lfloor (p_1+1)/2 \rfloor, \overline{n_1}]) \times [\underline{n_2}, \overline{n_2}] \cup \\ \cup [\underline{n_1}, \overline{n_1}] \times ([\underline{n_2}, \underline{n_2} + \lfloor (p_2+1)/2 \rfloor] \cup [\overline{n_2} - \lfloor (p_2+1)/2 \rfloor, \overline{n_2}]) \end{aligned}$$

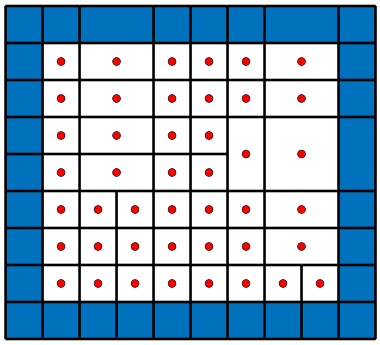
gültig sind. $AR_{\mathbf{p}}$ und $FR_{\mathbf{p}}$ für verschiedene $\mathbf{p}$ sind in Abbildung 3.1 zu sehen.

3.2. Zulässige Gitter, Anker

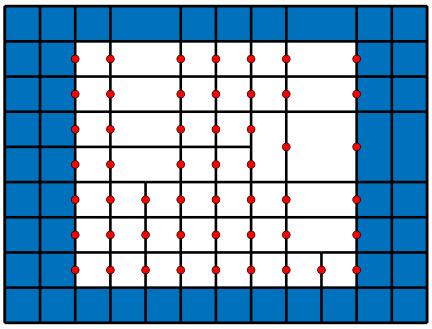
Definition 3.3 (Zulässige Gitter). *Ein T-Gitter $\mathcal{T}$ in der Index-Basis ist Element der Menge der zulässigen Gitter[9] $AD_{\mathbf{p}}$ für Polynomgrade p_1 und p_2, $\mathcal{T} \in AD_{\mathbf{p}}$, wenn $\mathcal{S} \cap FR_{\mathbf{p}}$ die Vertikalen*

$$\begin{aligned} \{\ell\} \times [\underline{n_2}, \overline{n_2}] \quad & \text{für} \quad \ell = \underline{n_1}, \underline{n_1}+1, \ldots, \underline{n_1}+p_1 \\ & \text{und} \quad \ell = \overline{n_1}-p_1, \overline{n_1}-p_1+1, \ldots, \overline{n_1}, \end{aligned}$$

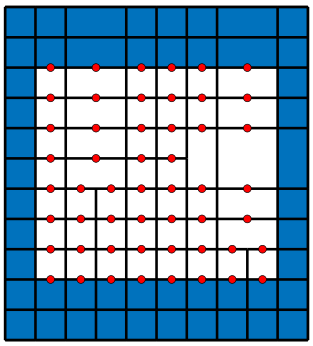
[5] Kanten, engl. Edges $E \in \mathcal{E}$
[6] Skelett, engl. Skeleton $S \in \mathcal{S}$
[7] Aktiver Bereich, engl. Active Region $\mathrm{AR}_{\mathbf{p}}$
[8] Randbereich, engl. Frame Region $\mathrm{FR}_{\mathbf{p}}$
[9] Zulässiges Gitter, engl. Admissible Mesh $\mathrm{AD}_{\mathbf{p}}$



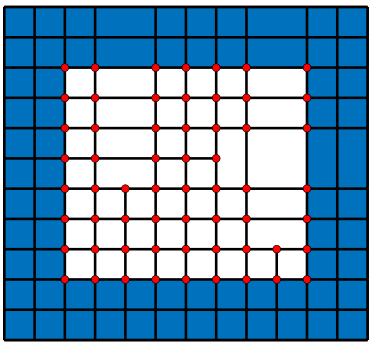
(a) $\mathbf{p} = (2, 2)$



(b) $\mathbf{p} = (3, 2)$



(c) $\mathbf{p} = (2, 3)$



(d) $\mathbf{p} = (3, 3)$

Abbildung 3.1.: Zulässige Gitter für verschiedene Polynomgrade $\mathbf{p} = (p_1, p_2)$. Der Randbereich ist blau hinterlegt. Eckpunkte als Anker sind durch einem Punkt darauf und Elemente, horizontale sowie vertikale Kanten sind als Anker durch einen Punkt in ihrer Mitte markiert.

und die Horizontalen

$$[\underline{n_1}, \overline{n_1}] \times \{\ell\} \quad \begin{array}{ll} \textit{für} & \ell = \underline{n_2}, \underline{n_2} + 1, \ldots, \underline{n_2} + p_2 \\ \textit{und} & \ell = \overline{n_2} - p_2, \overline{n_2} - p_2 + 1, \ldots, \overline{n_2}, \end{array}$$

enthält und alle $V \subset]\underline{n_1}, \overline{n_1}[\times]\underline{n_2}, \overline{n_2}[\cap FR_{\mathbf{p}}$ *Valenz vier haben.*

Diese Definition besitzt Ähnlichkeit zu jener des p-offenen Knotenvektors Ξ. Der Randbereich besteht auf beiden Seiten genau aus $p_\ell + 1$ Elementen, was ident mit der Zahl der Wiederholungen des Knotenvektors ist.

Definition 3.4 (Erweiterte Zulässigkeit von Gittern). *Ein T-Gitter* $\mathcal{T} \in AD_{\mathbf{p}}$ *gehört zu* $AD^+_{\mathbf{p}}$*, notiert durch* $\mathcal{T} \in AD^+_{\mathbf{p}}$*, wenn für alle Paare von Eckpunkten* $V^1 =$

$(i_1, j_1), V^2 = (i_2, j_2) \in \mathcal{V}$ *und* $V^1, V^2 \in \partial T$ *für ein existierendes* $T \in \mathcal{T}$ *sowie* $i_1 = i_2$ *bzw.* $j_1 = j_2$ *gilt, dass auch die offene Menge* $\{i_1\}\times]j_1, j_2[$ *bzw.* $]i_1, i_2[\times\{j_1\}$ *in* $\mathcal{S}$ *enthalten ist.*

Definition 3.5 (Anker). *Mit gegebenem T-Gitter* $\mathcal{T} \in AD_{\mathbf{p}}$ *wird die Menge der Anker* $\mathcal{A}_{\mathbf{p}}(\mathcal{T})$ *wie folgt definiert:*

- *wenn* p_1, p_2 *ungerade sind,* $\mathcal{A}_{\mathbf{p}}(\mathcal{T}) = \{\mathbf{A} \in \mathcal{V} : \mathbf{A} \subset AR_{\mathbf{p}}\}$,
- *wenn* p_1 *gerade und* p_2 *ungerade ist,* $\mathcal{A}_{\mathbf{p}}(\mathcal{T}) = \{\mathbf{A} \in \mathcal{E}^h : \mathbf{A} \subset AR_{\mathbf{p}}\}$,
- *wenn* p_1 *ungerade und* p_2 *gerade ist,* $\mathcal{A}_{\mathbf{p}}(\mathcal{T}) = \{\mathbf{A} \in \mathcal{E}^v : \mathbf{A} \subset AR_{\mathbf{p}}\}$,
- *wenn* p_1, p_2 *gerade sind,* $\mathcal{A}_{\mathbf{p}}(\mathcal{T}) = \{\mathbf{A} \in \mathcal{T} : \mathbf{A} \subset AR_{\mathbf{p}}\}$.

.

Die Menge der Anker für alle vier verschiedenen Kombinationen der Polynomgrade ist in Abbildung 3.1 zu sehen.

Definition 3.6 (Indexvektoren). *Für gegebenen Anker* $\mathbf{A} \in \mathcal{A}_{\mathbf{p}}$ *vom Typ* $a \times b$*, wobei* a *und* b *entweder Elemente von* $\mathbb{Z}$ *oder offene Intervalle mit ganzzahligen Endpunkten sind, werden der horizontale und der vertikale Indexvektor durch*

$$\mathcal{I}^h(a) := \{i \in \mathbb{Z} : \{i\} \times a \subset \mathcal{S}^v\},$$
$$\mathcal{I}^v(a) := \{j \in \mathbb{Z} : a \times \{j\} \subset \mathcal{S}^h\}$$

definiert und als geordnet angenommen.

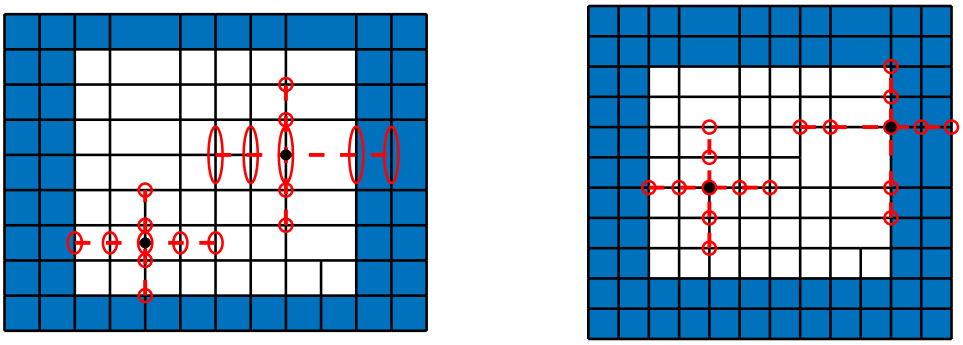
Definition 3.7 (Lokale Indexvektoren). *Mit gegebenem Anker* $\mathbf{A} = a \times b \in \mathcal{A}_{\mathbf{p}}(\mathcal{T})$*, wird der dazugehörige lokale horizontale bzw. vertikale Indexvektor* $v^h_{\mathbf{p}}(\mathbf{A})$ *bzw.* $v^v_{\mathbf{p}}(\mathbf{A})$ *als jene Teilmenge von* $\mathcal{I}^h(a)$ *bzw.* $\mathcal{I}^v(a)$ *definiert, die durch*

- p *ungerade:* $v^h_{\mathbf{p}}(\mathbf{A}) = (i_1, i_2, \ldots, i_{p_1+2}) \in \mathbb{Z}^{p_1+2}$ *besteht aus den eindeutigen* $p_1 + 2$ *aufeinanderfolgenden Indizes in* $\mathcal{I}^h(b)$*, sodass* $\{i_{(p_1+1)/2}\} = a$ *ist und*
- p *gerade:* $v^h_{\mathbf{p}}(\mathbf{A}) = (i_1, i_2, \ldots, i_{p_1+2}) \in \mathbb{Z}^{p_1+2}$ *besteht aus den eindeutigen* $p_1 + 2$ *aufeinanderfolgenden Indizes in* $\mathcal{I}^h(b)$*, sodass* $]i_{p_1/2}, i_{p_1/2+1}[= a$ *ist,*

gegeben ist. Der vertikale Indexvektor $v^v_{\mathbf{p}}(\mathbf{A}) = (j_1, j_2, \ldots, j_{p_2+2}) \in \mathbb{Z}^{p_2+2}$ *wird mit vertauschten* a *und* b *analog definiert.*

Beispiele für lokale Indexvektoren sind in Abbildung 3.2 gegeben.

Alle notwendigen Definitionen sind an an diesem Punkt getätigt, sodass schließlich die T-Spline-Kumulationsfunktionen zum T-Gitter $\mathcal{T}$ definiert werden können.



(a) $\mathbf{p} = (3, 2)$ (b) $\mathbf{p} = (3, 3)$

Abbildung 3.2.: Horizontale und vertikale lokale Indexvektoren für verschiedene $\mathbf{p} = (p_1, p_2)$. Die Anker sind mit schwarzen Punkten markiert. Die jeweils dazugehörigen lokalen Indexvektoren sind durch rote, gestrichelte Linien gekennzeichnet, die Elemente des Indexvektors durch rote Ellipsen.

Definition 3.8 (T-Spline-Kumulationsfunktionen). *Es seien im Intervall* $[0, 1]$ *der* p_1*-offene Knotenvektor* $\Xi_1 = \{\xi_{1,\underline{n_1}}, \xi_{1,\underline{n_1}+1}, \ldots, \xi_{1,\overline{n_1}}\}$ *und der* p_2*-offene Knotenvektor* $\Xi_2 = \{\xi_{2,\underline{n_2}}, \xi_{2,\underline{n_2}+1}, \ldots, \xi_{2,\overline{n_2}}\}$ *gegeben. Jeder Anker* $\mathbf{A} = a \times b \in \mathcal{A}_\mathbf{p}(\mathcal{T})$ *wird mit einer bivariaten B-Spline-Funktion nach Definition 2.8 vom Grad* $\mathbf{p} = (p_1, p_2)$ *verknüpft:*

$$B_{\mathbf{A},\mathbf{p}}(\zeta_1, \zeta_2) := B[\{\xi_{1,i_1}, \xi_{1,i_2}, \ldots, \xi_{1,i_{p_1+2}}\}](\zeta_1) \cdot B[\{\xi_{2,j_1}, \xi_{2,j_2}, \ldots, \xi_{2,j_{p_2+2}}\}](\zeta_2).$$

Hierbei sind $v^h_\mathbf{p}(\mathbf{A}) = (i_1, i_2, \ldots, i_{p_1+2})$ bzw. $v^v_\mathbf{p}(\mathbf{A}) = (j_1, j_2, \ldots, j_{p_2+2})$ der vertikale bzw. horizontale Indexvektor des Ankers $\mathbf{A}$. Der Raum der T-Splines ist dann definiert als jener Raum, der durch alle diese Kumulationsfunktionen aufgespannt wird, also

$$S^T_\mathbf{p}(\mathcal{A}_\mathbf{P}(\mathcal{T})) = \text{span}\{B_{\mathbf{A},\mathbf{p}} =, \mathbf{A} \in \mathcal{A}_\mathbf{p}(\mathcal{T})\}.$$

Der T-Spline-Raum ist identisch mit dem B-Spline-Raum $S_\mathbf{p}(\boldsymbol{\Xi})$, wenn das T-Gitter ein Tensorgitter ist. Eine T-Spline-Oberfläche wird analog zum multivariaten B-Spline-Fall definiert, nämlich indem jeder T-Spline-Funktion ein Kontrollpunkt im Raum zugewiesen wird.

Es lässt sich zeigen, dass die assoziierten Basisfunktionen $B_{\mathbf{A},\mathbf{p}}$ im Raum $S^T_\mathbf{p}(\mathcal{A}_\mathbf{P}(\mathcal{T}))$ auf allgemeinem T-Gitter **nicht** linear unabhängig sind (siehe hierfür [5, S.1437–1445]). Gelöst wird dieses Problem durch Einführen der Teilmenge der AST-Splines im folgenden Unterkapitel.

3.3. Analysis-suitable T-Splines

Ein Eckpunkt mit Valenz 3 wird auch T-Mündung[10] genannt. Je nach Orientierung wird dieser durch $\top, \bot, \vdash$ oder $\dashv$ notiert. Die T-Mündungen vom Typ $\top$ und $\bot$ bzw. $\vdash$ und $\dashv$ werden samt ihrer Erweiterung[11] vertikal bzw. horizontal genannt. Der Einfachheit wegen wird nur eine Mündung $M = \{(\bar{\imath}, \bar{\jmath})\}$ vom Typ $\dashv$ behandelt.

Definition 3.9 (T-Erweiterungen). *Es sei $\mathcal{I}^h(\{\bar{\jmath}\})$ der horizontalen Indexvektor einer Mündung $M = \{(\bar{\imath}, \bar{\jmath})\}$ vom Typ $\dashv$. Aus $\mathcal{I}^h(\{\bar{\jmath}\})$ werden genau jene $p+1$ aufeinanderfolgende Indizes $i_1, i_2, \ldots, i_{p_1+1}$ entnommen, für die*

$$\bar{\imath} = i_\kappa, \quad mit \quad \kappa = \lceil (p_1 + 2)/2 \rceil$$

gilt. Basierend darauf werden

- *die Kantenerweiterung*[12] $ext^k_{\mathbf{p}}(M) = [i_1, \bar{\imath}] \times \{\bar{\jmath}\}$,
- *die Frontalerweiterung*[13] $ext^f_{\mathbf{p}}(M) =]\bar{\imath}, i_{p_1+1}] \times \{\bar{\jmath}\}$
- *und schlicht die Erweiterung* $ext_{\mathbf{p}}(M) = ext^k_{\mathbf{p}}(M) \cup ext^f_{\mathbf{p}}(M)$,

der T-Mündung M definiert.

Beispiele für Kanten- und Frontalerweiterung sind in Abbildung 3.3 gegeben. Die horizontale Erweiterung der T-Mündung überspannt p_1 Indexintervalle, welche auch als Buchten[14] bezeichnet werden. Die Kantenerweiterung reicht über $\lceil (p_1 - 1)/2 \rceil$ und die Frontalerweiterung über $\lfloor (p_1 + 1)/2) \rfloor$ Buchten.

Definition 3.10 (Analysis-Suitability). *Ein T-Gitter $\mathcal{T} \in AD_{\mathbf{p}}$ ist AS*[15]*, wenn sich horizontale und vertikale Erweiterungen von T-Mündungen nicht schneiden.*

Beispielsweise ist das Gitter in Abbildung 3.3 AS und jenes in Abbildung 3.4 nicht-AS. Diese Klasse der Analysis-suitable T-Gitter wird durch $\mathrm{AS}_{\mathbf{p}}$ notiert.

Satz 3.11. *Unter der Voraussetzung, dass $\mathcal{T} \in AD^+_{\mathbf{p}} \cap AS_{\mathbf{p}}$, enthält der Raum der T-Splines $S^T_{\mathbf{p}}(\mathcal{A}_{\mathbf{P}}(\mathcal{T}))$ alle Polynome vom Grad p_1 in der ersten und p_2 in der zweiten Variable.*

Siehe [4] für einen Beweis von Satz 3.11.

Satz 3.12. *Sei $\mathcal{T} \in AD^+_{\mathbf{p}} \cap AS_{\mathbf{p}}$, dann sind die T-Spline-Kumulationsfunktionen nach Definition 3.8*

[10] T-Mündung, engl. T-junction M
[11] Erweiterung, engl. Extension $\mathrm{ext}_{\mathbf{p}}(M)$
[12] Kantenerweiterung, engl. Edge extension $\mathrm{ext}^k_{\mathbf{p}}(M)$
[13] Frontalerweiterung, engl. Face extension $\mathrm{ext}^f_{\mathbf{p}}(M)$
[14] Buchten, engl. Bays
[15] Analysis-suitable (AS)

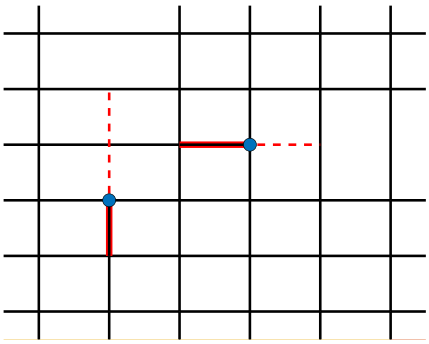


Abbildung 3.3.: T-Erweiterungen für Polynomgrade $p_1 = 2$ (horizontal) und $p_2 = 3$ (vertikal): Die Frontalerweiterung ist gestrichelt markiert

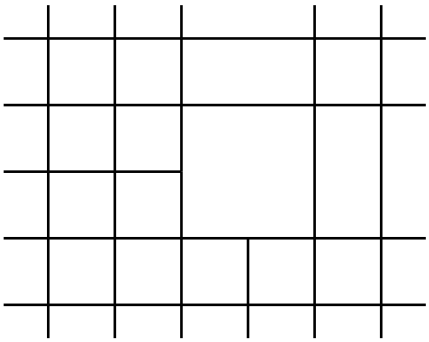


Abbildung 3.4.: Ein Gitter $\mathcal{T}$ mit $\mathbf{p} = (3, 2)$. Es gilt $\mathcal{T} \notin \mathrm{AS}_{\mathbf{p}}$ nach Definition 3.10, da sich die Erweiterungen der T-Mündungen schneiden.

(i). linear unabhängig und

(ii). eine Zerlegung der Eins.

Für einen Beweis von Satz 3.12 und weitere Eigenschaften von AST-Splines siehe [2, S. 255 f.]. Die AS-Eigenschaft bleibt bei allgemeinen Verfeinerungen nicht erhalten (z.B. der Gitterausschnitt aus Abbildung 3.4 ist leicht konstruierbar). Abhilfe schafft hier der Verfeinerungsalgorithmus im folgenden Unterkapitel.

3.4. Lokale dyadische Verfeinerung

Der folgende Abschnitt ist aus [9] entnommen. Im Morgenstern-Peterseim-Algorithmus wird zusätzlich zur gegebenen, initial zu verfeinernden Menge eine weitere Menge von

problematischen Nachbaren verfeinert. Die Auswahl dieser erfolgt aufgrund eines Distanzkriteriums. Es können damit mehrere Elemente pro Durchlauf verfeinert werden und es wird die AS-Eigenschaft des Gitters und damit die lineare Unabhängigkeit der assoziierten Basisfunktionen erhalten. Außerdem ist der Algorithmus auf höhere Dimensionen erweiterbar.

Definition 3.13 (Uniformes T-Gitterlevel). *Für gegebenes Level $k \in \mathbb{N}$ wird das uniforme T-Gitter $\mathcal{T}_{u[k]}$ definiert durch*

$$\mathcal{T}_{u[k]} := \left\{[x - 2^{-k}, x] \times [y - 2^{-k}, y] : 2^k x \in \{1, 2, \ldots, 2^k\}, 2^k y \in \{1, 2, \ldots, 2^k\}\right\}.$$

Das Gitter aus Definition 3.13 überdeckt die Parameter-Domäne $\Omega = [0, 1]^2$ und alle Elemente haben paarweise disjunktes Inneres.

Definition 3.14 (Uniforme Übergangs-Gitter). *Für gegebenes Level k wird das uniformen Übergangs-Gitter[16] $\mathcal{T}_{u[k+1/2]}$ definiert durch*

$$\mathcal{T}_{u[k+1/2]} := \left\{[x - 2^{-k-1}, x] \times [y - 2^{-k}, y] : 2^{k+1} x \in \{1, 2, \ldots, 2^{k+1}\}, \right. \\ \left. 2^k y \in \{1, 2, \ldots, 2^k\}\right\}.$$

Definition 3.15 (Menge der uniformen Gitter). *Die Menge aller uniformen T-Gitter wird mit*

$$\mathbb{T}_U := \left\{\mathcal{T}_{u[k]} : 2k \in \mathbb{N}\right\}$$

bezeichnet.

Definition 3.16 (Allgemeines T-Gitter). *Ein allgemeines T-Gitter besteht aus endlich vielen Elementen aus $\bigcup_{\mathcal{T}' \in \mathbb{T}_U} \mathcal{T}'$, sodass alle Elemente paarweise disjunktes Inneres haben und die Vereinigung aller Elemente die Parameter-Domäne $\Omega = [0, 1]^2$ überdeckt:*

$$\mathbb{T} := \Big\{\mathcal{T} \subset \bigcup_{\mathcal{T}' \in \mathbb{T}_U} \mathcal{T}' : \#\mathcal{T} < \infty, \bigcup \mathcal{T} = [0, 1] \times [0, 1], \\ \forall\, T \in \mathcal{T}, \forall\, T' \in \mathcal{T} \backslash \{T\} : \ int(T) \cap \ int(T') = \emptyset\Big\}$$

Definition 3.17 (Element-Level). *Zu jedem Element $T \in \mathcal{T}$ existiert ein eindeutiges k, $2k \in \mathbb{N}$, mit $T \in \mathcal{T}_{u[k]}$. Das Level von T ist dann definiert durch*

$$\ell(T) := k.$$

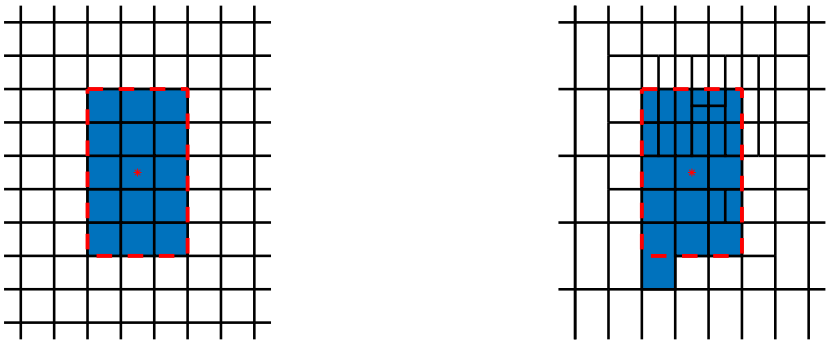


Abbildung 3.5.: Nachbarschaft des mit rotem Stern markierten Elementes T für uniformes (links) und nicht-uniformes Gitter (rechts). Blau hinterlegt ist die Nachbarschaft erster Ordnung, rot strichliert der Bereich, der vektorwertig näher als $\mathbf{D}(k)$ nach Definition 3.21 liegt. Das Level $\ell(T)$ ist ganzzahlig, der Polynomgrad $\mathbf{p} = (3, 3)$.

Definition 3.18 (Prozedur Teile(T)). *Für gegebenes T-Gitter $\mathcal{T}$ und Element $T \in \mathcal{T}_{u[k]}$ mit $2k \in \mathbb{N}$ ist die Prozedur zur Verfeinerung definiert durch*

$$\textsc{Teile}(T) := \left\{T' \in \mathcal{T}_{u[k+1/2]} : T' \subset T\right\}.$$

Definition 3.19 (Prozedur Teile($\mathcal{T}, \mathcal{M}$)). *Für gegebenes T-Gitter $\mathcal{T}$ und eine Menge von markierten Elementen $\mathcal{M} \subseteq \mathcal{T}$ setzt sich die Verfeinerung von $\mathcal{M}$ aus der Verfeinerung der einzelnen Elemente zusammen:*

$$\textsc{Teile}(\mathcal{T}, \mathcal{M}) := \mathcal{T} \setminus \mathcal{M} \cup \bigcup_{T \in \mathcal{M}} \textsc{Teile}(T).$$

Die Prozedur in Definition 3.19 erhält aber die AS-Eigenschaft für allgemeine markierte Mengen nicht. Für den zentralen Algorithmus müssen noch weitere Begriffe eingeführt werden. Dazu sei $\operatorname{abs}(z) := (|z_j|)_{j=1,2} \in \mathbb{R}^2$ für $z \in \mathbb{R}^2$ der komponentenweise Absolutbetrag.

[16] Intermediate uniform meshes

Definition 3.20 (Vektorwertiger Abstand). *Für gegebenen Punkt $z := (x, y) \in \Omega = [0,1]^2$ und Element $T \in \mathcal{T}$ wird der vektorwertige Abstand als Absolutwert der komponentenweisen Differenz zwischen (x, y) und dem Mittelpunkt von T definiert:*

$$\mathit{Dist}(T, z) := \ \mathit{abs}\big(\ \mathit{mid}(T) - (x, y)\big) \ \in \mathbb{R}^2.$$

Für den Abstand zweier Elemente $T, T' \in \mathcal{T}$ wird die kompakte Notation

$$\mathit{Dist}(T, T') := \ \mathit{abs}\big(\ \mathit{mid}(T) - \ \mathit{mid}(T')) \ \in \mathbb{R}^2$$

verwendet.

Definition 3.21 (Gröbere Nachbarschaft). *Für $T \in \mathcal{T}$ mit $\ell(T) > 0$ ist die gröbere Nachbarschaft definiert durch*

$$\mathcal{N}(\mathcal{T}, T) := \big\{T' \in \mathcal{T} \cap \mathcal{T}_{u[\ell(T)-1/2]} : \ \mathit{Dist}(T, T') \leq \mathbf{D}(\ell(T))\big\},$$

mit

$$\mathbf{D}(k) = \begin{cases} 2^{-k}(\lfloor \frac{p_1}{2} \rfloor + \frac{1}{2}, \lceil \frac{p_2}{2} \rceil + \frac{1}{2}) & \text{für} \quad k \in \mathbb{N}, \\ 2^{-k-1/2}(\lceil \frac{p_1}{2} \rceil + \frac{1}{2}, 2\lfloor \frac{p_2}{2} \rfloor + 1) & \text{sonst}, \end{cases}$$

wobei $\mathbf{p} = (p_1, p_2)$ der Polynomgrad in x- bzw. y-Richtung ist. Die gröbere Nachbarschaft für eine Menge von markierten Elementen $\mathcal{M} \subseteq \mathcal{T}$ ist definiert durch

$$\mathcal{N}(\mathcal{T}, \mathcal{M}) := \bigcup_{T \in \mathcal{M}} \mathcal{N}(\mathcal{T}, T).$$

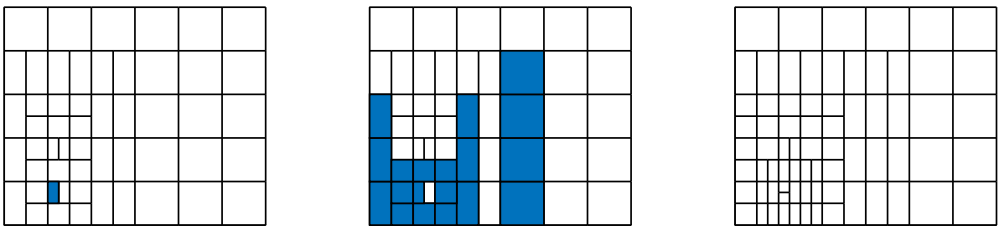
Definition 3.22 (Abschluss$(\mathcal{T}, \mathcal{M})$). *Für gegebenes T-Gitter $\mathcal{T}$ und eine Menge von markierten Elementen $\mathcal{M} \subseteq \mathcal{T}$ ist der Abschluss von $\mathcal{M}$ definiert durch*

$$\text{Abschluss}(\mathcal{T}, \mathcal{M}) := \bigcup_{k=0}^{2 \cdot \ max \ \ell(\mathcal{M})} \mathcal{N}^k(\mathcal{T}, \mathcal{M})$$

mit der groben Nachbarschaft k-ter Ordnung

$$\mathcal{N}^k(\mathcal{T}, \mathcal{M}) := \begin{cases} \mathcal{M} & \text{für} \quad k = 0, \\ \mathcal{N}(\mathcal{T}, \mathcal{N}^{k-1}(\mathcal{T}, \mathcal{M})) & \text{sonst.} \end{cases}$$

Die Nachbarschaft für je ein uniformes Gitter und ein nicht-uniformes Gitter sind in Abbildung 3.5 zu sehen.



(a) Initial markiertes Element und initiales Gitter (b) Abschluss des markierten Elements (c) Verfeinertes Gitter

Abbildung 3.6.: Visualisierung des Verfeinerungsprozesses VERFEINERE$(\mathcal{T}, \mathcal{M})$ aus Definition 3.23 für ein einzelnes markiertes Element für Polynomgrad $\mathbf{p} = (3, 3)$.

Definition 3.23 (VERFEINERE$(\mathcal{T}, \mathcal{M})$). *Für gegebenes T-Gitter $\mathcal{T}$ und eine Menge von markierten Elementen $\mathcal{M} \subseteq \mathcal{T}$ ist der erweiterte Algorithmus für Verfeinerung durch*

$$\text{VERFEINERE}(\mathcal{T}, \mathcal{M}) := \text{TEILE}\big(T, \text{ABSCHLUSS}(\mathcal{Q}, \mathcal{M})\big)$$

definiert.

Ein Beispiel für den erweiterten Algorithmus aus Definition 3.23 zur Verfeinerung von T-Gittern VERFEINERE$(\mathcal{T}, \mathcal{M})$ ist in Abbildung 3.6 zu sehen.

Algorithmus 3.24 (T-GITTER-VERFEINERUNG).

Input: *T-Gitter $\mathcal{T}$, Menge von markierten Elementen $\mathcal{M}_{init} \subseteq \mathcal{T}$*

(i). Setze `maxLevel` $:=$ $max\ell(\mathcal{M})$ *und definiere* $\mathcal{M}_{result} := \mathcal{M}_{init}$

(ii). ***Schleife:*** *Für $i = 0, 1, \ldots, 2 \cdot max\ell(\mathcal{M})$ mache:*

$\mathcal{M}_{new} := \emptyset$

Schleife: *Für $j = 1, 2, \ldots, \#\mathcal{M}_{init}$ mache:*

$\mathcal{M}_{new} := \mathcal{M}_{new} \cup \mathcal{N}(\mathcal{T}, \mathcal{M}_{init}[j])$

$\mathcal{M}_{init} := \mathcal{M}_{new} \backslash \mathcal{M}_{result}$

Wenn $\mathcal{M}_{init} \neq \emptyset$ *dann setze* $\mathcal{M}_{final} := \mathcal{M}_{final} \cup \mathcal{M}_{new}$

sonst *brich die Schleife ab und setze* $\mathcal{T}' :=$ TEILE $(\mathcal{T}, \mathcal{M}_{final})$

Output: *Verfeinertes T-Gitter $\mathcal{T}'$*

Satz 3.25. *Für T-Gitter $\mathcal{T} \in AS_{\mathbf{p}}$ bzw. $\mathcal{T} \in AD_{\mathbf{p}^+}$ und eine Menge von markierten Elementen $\mathcal{M} \subseteq \mathcal{T}$ gilt für das verfeinerte T-Gitter $\mathcal{T}' :=$ VERFEINERE$(\mathcal{T}, \mathcal{M})$ ebenfalls $\mathcal{T}' \in AS_{\mathbf{p}}$ bzw. $\mathcal{T}' \in AD_{\mathbf{p}^+}$.*

Ein Beweis für Satz 3.25 ist in [9, S. 50, ff] zu finden.

4. Adaptive isogeometrische Methode

In dieser Arbeit wird eine FEM[1] verwendet, das ist eine numerische Methode zur approximativen Lösung von PDEs[2] basierend auf dem Galerkin-Verfahren[3]. Für die (Gitter-) adaptive FEM wurde in Kombination mit der Dörfler-Markierungsstrategie[4] 2004 in [3] ein erster Beweis für optimale Konvergenz gegeben und 2014 allgemeiner für eine sehr breite Variation an Problemen in [6].

IGA[5] ist ein Begriff für eine Menge von Galerkin-Methoden, die 2005 in [8, 7] eingeführt wurden, um das Konzept von CAD/CAM[6] zu revolutionieren. Die adaptive IGAFEM, also adaptive FEM im Rahmen von IGA, ist ein iterativer Algorithmus mit vier Schritten:

$$\text{Lösen} \to \text{Abschätzen} \to \text{Markieren} \to \text{Verfeinern}$$

4.1. Poisson-Problem

Definition 4.1 (Poisson-Gleichung). *Für gegebene Funktion $f(x, y)$ auf dem Gebiet $\Omega = (0, 1)^2$ ist durch die Differentialgleichung*

$$\begin{aligned} -\Delta u(x, y) &= f(x, y) && \text{auf } \Omega, \\ u(x, y) &= 0 && \text{auf } \partial\Omega \end{aligned}$$

die Poisson[7]-Gleichung mit homogenen Dirichlet[8]-Randbedingungen gegeben.

Gleichungen nach Definition 4.1 sind von essentieller Bedeutung für viele Bereiche der Physik. In der Elektrodynamik ist der Zusammenhang von elektrischem Potential $\varphi_{e\ell}$ und der Ladungsverteilung $\rho_{e\ell}$ durch eine Gleichung der Form

$$-\Delta\varphi_{e\ell} = \rho_{e\ell}.$$

[1] Finite-Elemente-Methode (FEM)
[2] Partielle Differentialgleichungen (PDE)
[3] Boris Grigoryevich Galerkin, geb. 1871 in Polotsk, damals Russland, gest. 1945 in Moskau
[4] Willy Dörfler
[5] Isogeometrische Analysis (IGA)
[6] Computer-aided Design (CAD), Computer-aided Manufacturing (CAM)
[7] Siméon Denis Poisson, geb. 1781 in Pithiviers (Département Loiret), gest. 1840 in Paris
[8] Johann Peter Gustav Lejeune Dirichlet, geb. 1805 in Düren, gest. 1859 in Göttingen

gegeben (unter Vernachlässigung von Konstanten im SI-System). Für die Massendichte ρ_m und das Gravitationspotential φ_m ergibt sich ein identischer Zusammenhang, ebenso innerhalb der Strömungslehre mit der Quellendichte ρ_q und dem Strömungspotential φ_q.

Mathematisch betrachtet, handelt es sich beim Poisson-Problem um eine elliptische Differenzialgleichung zweiter Ordnung. Durch partielle Integration nach Multiplikation mit einer Funktion $v \in H_0^1(\Omega)$ im Hilbert-Raum erhält man durch Lösen der Variationsformulierung aus Definition 4.2.

Definition 4.2 (Variationsformulierung). *Für gegebene Funktion $f(x, y)$ ist die Variationsformulierung der Poisson-Gleichung definiert durch*

$$\int_\Omega \nabla u \cdot \nabla v \, d(x, y) = \int_\Omega f \, v \, d(x, y) \quad \textit{für alle } v \in H_0^1(\Omega),$$

wobei $u, v \in H_0^1(\Omega)$ sind. Die Unbekannte u ist dabei die sogenannte schwache Lösung.

Durch die FEM wird das Problem diskretisiert, für einen endlichdimensionalen Unterraum $U_h \subset H_0^1(\Omega)$ mit gegebener Basis $\{B_1, \ldots, B_{n_h}\}$ wird die schwache Formulierung zu einer diskreten Variationsformulierung.

Definition 4.3 (Diskrete Variationsformulierung). *Die diskrete Variante der Variationsformulierung lautet für gegebene Funktion $f(x, y)$ wie folgt: Finde eine Funktion $u_h \in U_h \subset H_0^1$, für die gilt*

$$\langle\!\langle u_h, v_h \rangle\!\rangle = F(v_h) \quad \textit{für alle } v_h \in U_h,$$

wobei die Abkürzungen

$$\langle\!\langle u_h, v_h \rangle\!\rangle := \int_\Omega \nabla u_h \cdot \nabla v_h \, d(x, y), \quad F(v_h) := \int_\Omega f \, v_h \, d(x, y)$$

verwendet wurden.

Die Mathematik zeigt, dass sowohl Definition 4.2 als auch Definition 4.3 eindeutige Lösungen $u \in H_0^1(\Omega)$ bzw. $u_h \in U_h$ haben. Ferner ist die sogenannte Galerkin-Approximation die Bestapproximation von u in dem Sinn, dass $\|u - u_h\| = \min_{u_h \in U_h} \|u - u_h\|$, wobei $\|u\| := \sqrt{\langle\!\langle u, u \rangle\!\rangle}$. Da $u_h \in U_h$, existieren eindeutige Koeffizienten x_j mit

$$u_h = \sum_{j=1}^{n_h} x_j B_j.$$

Die Koeffizienten x_j müssen allerdings noch bestimmt werden. Mit dem Ansatz $v_h = \sum_{i=1}^{n_h} y_i B_i$ wird das innere Produkt zu

$$\langle\!\langle u_h, v_h \rangle\!\rangle = \sum_{i,j=1}^{n_h} \langle\!\langle \nabla(y_i B_i), \nabla(x_j B_j) \rangle\!\rangle = \sum_{i,j=1}^{n_h} y_i x_j \underbrace{\langle\!\langle \nabla B_i, \nabla B_j \rangle\!\rangle}_{:= A_{ij}} = \sum_{i,j=1}^{n_h} y_i x_j A_{ij}.$$

Für die rechte Seite ergibt sich

$$F(v_h) = \sum_{i=1}^{n_h} y_i \underbrace{F(B_i)}_{:=F_i} = \sum_{i=1}^{n_h} y_i F_i.$$

Die resultierende Gleichung

$$\sum_{i,j=1}^{n_h} y_i x_j A_{ij} = \sum_{i=1}^{n_h} y_i F_i$$

ist für alle y_i gültig und damit folgt für die Matrix $\mathbf{A} = (A_{ij})_{i,j=1,\dots,n_h}$ und die Vektoren $\mathbf{F} = (F_i)_{i=1,\dots,n_h}$ und $\mathbf{x} = (x_j)_{j=1,\dots,n_h}$, dass

$$\mathbf{F} = \mathbf{A}\mathbf{x}.$$

Wegen der Koerzivität der Bilinearform $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ ist die Matrix $\mathbf{A}$ positiv definit und deswegen invertierbar. Damit gilt

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{F},$$

d.h. die Bestapproximation u_h von u in U_h ist numerisch berechenbar.

4.2. Adaptive IGAFEM

Der adaptive IGAFEM-Algorithmus (Algorithmus 4.4) wird in einer Schleife aus vier Teilschritten durchgeführt, wobei als Basis die Menge aller assoziierten Basisfunktionen des T-Spline-Gitters verwendet wird. Eine Implementierung in MATLAB ist in Kapitel 5.2 zu finden.

Algorithmus 4.4 (ADAPTIVE IGAFEM).

Input: *Gitter $\mathcal{T}_0 = \mathcal{T}$, Funktion $f(x, y)$, Verfeinerungsparameter $0 < \theta \leq 1$, Zähler $\ell := 0$*

(i). Berechne die Galerkin-Lösung $u_\ell \in U_\ell := S_0^1(\mathcal{T}_\ell)$.

(ii). Berechne Verfeinerungsindikatoren $\eta_\ell(T)$ für alle $T \in \mathcal{T}_\ell$.

(iii). Stop, wenn $\eta_\ell^2 = \sum_{T\in\mathcal{T}_\ell} \eta_\ell^2(T)$ hinreichend klein.

(iv). Anderfalls bestimme die minimale Menge $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell$, für die gilt

$$\theta \sum_{T\in\mathcal{T}_\ell} \eta_T^2 \leq \sum_{T\in\mathcal{M}_\ell} \eta_T^2.$$

(v). Erzeuge neues Gitter $\mathcal{T}_{\ell+1} :=$ VERFEINERE$(\mathcal{T}_\ell, \mathcal{M}_\ell)$

(vi). Erhöhe den Zähler $\ell \mapsto \ell + 1$ und starte wieder bei (i).

Output: *Verfeinertes Gitter $\mathcal{T}_\ell$, zugehörige Lösung u_h und Fehlerschätzer η_ℓ*

4.2.1. Lösen

Die Grundlagen für diesen Teil wurden bereits im vorherigen Unterkapitel erklärt. Ziel ist es, für gegebenes T-Spline-Gitter $\mathcal{T}_\ell$ und damit gegebene Basis B_i die Lösung u_h zu finden. Die wichtigen Resultate und Definitionen sind

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{F}, \quad u_h = \sum_{j=1}^{n_h} x_j B_j, \quad A_{i,j} := \langle\!\langle \nabla B_i, \nabla B_j \rangle\!\rangle \quad \text{und} \quad F_i := F(B_i).$$

Zusammen mit der Definition für das innere Produkt $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ und das Funktional $F(\cdot)$ werden Matrix $\mathbf{A}$ und Vektor $\mathbf{F}$ auf Ω zu

$$\mathbf{A}_{ij} = \int_\Omega \nabla B_i \cdot \nabla B_j d(x,y) \quad \text{und} \quad \mathbf{F}_i = \int_\Omega f B_i d(x,y).$$

4.2.2. Abschätzen

Zweck des A-Posteriori-Fehlerschätzers ist es, für bekannte Galerkin-Lösung u_ℓ den Fehler jedes Elements $T \in \mathcal{T}_\ell$ abzuschätzen.

Definition 4.5 (Fehlerindikator). *Für gegebene Funktion $f(x,y)$, Element $T \in \mathcal{T}_\ell$ und Galerkin-Lösung $u_\ell \in U_h \subset H_0^1(\Omega)$ ist der Fehlerindikator η_T gegeben durch*

$$\eta_T^2 = |T| \|f + \Delta u_\ell\|_T^2.$$

Satz 4.6. *Für gegebene Galerkin-Lösung $u_\ell \in U_h \subset H_0^1(\Omega)$ und T-Gitter $\mathcal{T}_\ell$ gilt*

$$\|u - u_\ell\| \leq C \left(\sum_{T \in \mathcal{T}_\ell} \eta_T^2 \right)^{1/2},$$

wobei $C > 0$ nur von der zugrundeliegenden stetigen und elliptischen Bilinearform $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ abhängt.

Siehe [10] für einen Beweis von Satz 4.6.

4.2.3. Markieren

Verwendet wird die Markierungs-Strategie von Dörfler: Bestimme für festem Adaptivitätsparameter $\theta \in (0, 1]$ die minimale Menge $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell$, für die gilt:

$$\theta \sum_{T \in \mathcal{T}_\ell} \eta_T^2 \leq \sum_{T \in \mathcal{M}_\ell} \eta_T^2.$$

Generisch korrespondiert ein kleines $0 < \theta \ll 1$ zu weniger markierten Elementen und damit zu hochgradig adaptierten Gittern, während $\theta = 1$ der Markierung aller Elemente und damit uniformer Verfeinerung entspricht.

4.2.4. Verfeinern

Die soeben markierte Menge $\mathcal{M}_\ell$ wird als Initialmenge für den Verfeinerungsalgorithmus von Morgenstern–Peterseim verwendet (Definition 3.23. Die resultierende Menge wird verfeinert und das neue Gitter ist der Ausgangspunkt für den nächsten Durchlauf mit $\mathcal{T}_{\ell+1} = \textsc{Verfeinere}(\mathcal{T}_\ell, \mathcal{M})$.

5. Implementierung in Matlab

Zentrales Anliegen der vorliegenden Bachelorarbeit ist die objektorientierte Implementierung von IGAFEM mit T-Splines in MATLAB. Im ersten Unterkapitel werden Speicherung und Manipulation allgemeiner T-Gitter beschrieben, im zweiten die effiziente Durchführung der adaptiven IGAFEM. Ein Überblick über alle Methoden der Klassen ist in den entsprechenden Listings in den Kapiteln zu finden, der vollständige Quellcode im Anhang.

5.1. Implementierung des Gitters $\mathcal{T}$

Die Klasse in der Quellcode-Datei `TMesh.m` ist die Repräsentation eines T-Gitters $\mathcal{T}$ mit dazugehörigen Methoden, unter anderem zur Gitterverfeinerung, Ankerberechnung und zur visuellen Ausgabe, siehe Listing 5.1. Sie enthält die Instanzvariablen

- `knotsXGlobal`, der globale, geordnete Knotenvektor $\Xi_1 := (\underline{n_1}, \underline{n_1}+1, \ldots, \overline{n_1})$ in seiner Parameterdarstellung,
- `knotsYGlobal`, der globale, geordnete Knotenvektor $\Xi_2 := (\underline{n_2}, \underline{n_2}+1, \ldots, \overline{n_2})$ in seiner Parameterdarstellung,
- `vertices`, eine Matrixrepräsentation der Menge der Eckpunkte, $\mathcal{V}$,
- `vEdges`, eine Matrixrepräsentation der Menge der vertikalen Kanten, $\mathcal{E}^v$,
- `hEdges`, eine Matrixrepräsentation der Menge der horizontalen Kanten, $\mathcal{E}^h$,
- `tiles`, eine Tensor-Repräsentation des T-Gitters $\mathcal{T}$,

siehe Abbildung 5.1 für ein Beispiel. Der formale Aufbau der Datenstruktur folgt in Definition 5.4. Der Konstruktor

- `TMesh(initLevel, initDegree)` hat als erstes Argument das Level ℓ_{init} des anfänglichen uniformen Gitters und den Vektor der Polynomgrade $\mathbf{p} = (p_1, p_2)$. Der vollständige Code ist in Listing A.1 zu finden.

Die öffentlichen Methoden der Klasse sind:

- `getAnchors(obj)` zur Berechnung und Rückgabe der Koordinaten aller Anker $A_i = (x_i, y_i) \in \mathcal{A}_{\mathbf{p}}(\mathcal{T})$ und dazugehöriger lokaler Knotenvektoren $v^v_{\mathbf{p}}(A_i)$ bzw. $v^h_{\mathbf{p}}(A_i)$. Siehe Listing A.2 für dessen Implementierung und Unterkapitel 5.1.2 für Details.

- `getUniqueGlobalKnots(obj)` gibt die globalen Knotenvektoren ohne Wiederholungen, Z_1 und Z_2, berechnet aus `knotsXGlobal` und `knotsYGlobal` (Listing A.3).
- `getVertices(obj)` retourniert eine Liste aller Eckpunkte $E \in \mathcal{E}$ in der Parameterbasis (Listing A.4). Sie wird benötigt für die Berechung des punktweisen Fehlers.
- `getMaxTileLevel(obj)` gibt $\max\{\ell(T)\}_{T\in\mathcal{T}}$ zurück (Listing A.5).
- `getActiveTiles(obj)` gibt die Koordinaten der Eckpunkte (x_1, y_1, x_2, y_2) aller Elemente $T = \big((x_1, y_1), (x_2, y_1), (x_2, y_2), (x_1, y_2)\big) \in \mathcal{T}$ zurück, deren Eckpunkte alle im aktiven Bereich, $\mathrm{AR}_{\mathbf{p}}$ nach Definition 3.2 liegen (Listing A.6).
- `getNonzeroActiveTiles(obj)` entfernt aus `getActiveTiles(obj)` noch jene Elemente $T \in \mathcal{T}$, für die $(x_1 = x_2) \vee (y_1 = y_2)$ gilt, also Fläche null haben (Listing A.7).
- `refineTile(obj, toRefine)` gibt das nach dem Algorithmus 3.24 von Morgenstern verfeinerte TGitter $\mathcal{T}'$ als `TMesh`-Objekt zurück sowie die Mächtigkeit des Abschlusses von $\mathcal{M}_{init}$ nach Definition 3.22, also die Anzahl der Elemente $\mathcal{M}_{final}$, die verfeinert wurden. Das Argument `toRefine` gibt dabei die Menge der markierten Elemente $\mathcal{M}_{init} \subseteq \mathcal{T}$ in der Parameter-Basis an (Listing A.8). Dieses wird dann, in die Indexbasis dargestellt an die private Tochterfunktion `morgensternRefine(obj, initialTiles)` übergeben werden. Die Rückgabe-Parameter dieser Tochterfunktion werden direkt als Rückgabe dieser Funktion übernommen. Details dazu in Unterkapitel 5.1.2.

Listing 5.1.

```
1  classdef TMesh < handle
2  % TMesh: container of vertices, edges and knot vectors
3  % with methods for refinement and plotting
4
5  properties
6    knotsXGlobal
7    knotsYGlobal
8    vertices
9    vEdges
10   hEdges
11   tiles
12   initialTileSize
13   degree
14 end
15
16 methods
17   function obj = TMesh(initLevel, deg)
18
19   function [anchors, localX, localY] = getAnchors(obj)
20
21   function [xGlobal, yGlobal] = getUniqueGlobalKnots(obj)
22
23   function [xValues, yValues] = getVertices(obj)
```

```
24
25    function maxTileLevel = getMaxTileLevel(obj)
26
27    function result = getActiveTiles(obj)
28
29    function result = getNonzeroActiveTiles(obj)
30
31    function [obj, numRefined] = refineTiles(obj, toRefine)
32  end
```

5.1.1. Speicherung des T-Gitters

Die Eckpunkte, Kanten und Elemente des Gitters werden in den dazugehörigen Matrizen bzw. Tensoren

$$\texttt{vertices}, \texttt{vEdges}, \texttt{hEdges} \in \mathbb{Z}^{\texttt{xDim}\times\texttt{yDim}} \quad \text{bzw.} \quad \texttt{tiles} \in \mathbb{Z}^{\texttt{xDim}\times\texttt{yDim}\times 3}$$

gespeichert, unter Verwendung der Abkürzungen `xDim` := length(`knotsXGlobal`) sowie `yDim` := length(`knotsYGlobal`). Jedem möglichen Gitterpunkt entspricht dabei ein Eintrag in der Matrix.

Definition 5.2 (BASIS-ECKPUNKT). *Für gegebenen Index-Bereich $I = [i_1, i_2] \times [j_1, j_2]$ mit den Eckpunkten $(i_1, j_1), (i_2, j_2) \in \mathcal{V}$ sowie $i_1 \leq i_2$ und $j_1 \leq j_2$ ist der Basis-Eckpunkt $V_B(I) = (i_B, j_B) \in \mathcal{V}$ gegeben durch*

$$V_B(I) := (i_1, j_1),$$

Für die Menge der Basis-Eckpunkte einer Menge von Index-Teilbereichen $\mathcal{I} = \{I_\ell\}$ wird die Abkürzung

$$\mathcal{V}_B(\mathcal{I}) := \bigcup_{I\in\mathcal{I}} V_B(I)$$

verwendet.

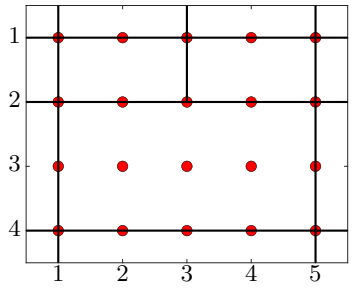
Jeder Basis-Eckpunkt in den Mengen $\mathcal{V}_B(\mathcal{E}^v)$, $\mathcal{V}_B(\mathcal{E}^h)$ bzw. $\mathcal{V}_B(\mathcal{T})$ hat eine eindeutige Kante in $\mathcal{E}^v$, $\mathcal{E}^h$ bzw. ein Element in $\mathcal{T}$.

Definition 5.3 (BASIS-KANTENPUNKTE). *Für gegebenes Element $T = [i_1, i_2] \times [j_1, j_2] \in \mathcal{T}$ ist die Menge der Basis-Kantenpunkte $\mathcal{B}(T) \subset \partial T$ gegeben durch*

$$\mathcal{B}(T) := \{P = (i, j) : (i_1 < i < i_2 \wedge j = j_1) \vee (i = i_1 \wedge j_1 < j < j_2)\}$$

Die Einträge der Matrizen bzw. Tensoren entsprechen folgendem System:

- `vertices` ist eine Wahrheitsmatrix, ob sich am Punkt (i, j) ein Eckpunkt befindet.



(a) 4×5-Ausschnitt eines Gitters in der Index-Basis

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) `vertices`

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(c) `vEdges`

$$\begin{pmatrix} 2 & -1 & 2 & -1 & 1 \\ 2 & -1 & 2 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & -1 & -2 & -3 & 2 \end{pmatrix}$$

(d) `hEdges`

$$\begin{pmatrix} 2 & -1 & 2 & -1 & 1 \\ 4 & -1 & -2 & -3 & 2 \\ 0 & -1 & -2 & -3 & 0 \\ 4 & -1 & -2 & -3 & 2 \end{pmatrix}$$

(e) `tiles`(:,:,1)

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 2 \\ -1 & -1 & -1 & -1 & -1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(f) `tiles`(:,:,2)

$$\begin{pmatrix} 3.5 & 0 & 3.5 & 0 & 3.5 \\ 3 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 3 \end{pmatrix}$$

(g) `tiles`(:,:,3)

Abbildung 5.1.: Datenstruktur des T-Gitters. Jedem der 4×5 Gitterpunkte (durch rote Punkte markiert) des Gitters (5.1a) entspricht je ein Eintrag in allen 4×5-Matrizen (5.1b) - (5.1g).

- `vEdges` hat an der Stelle (i, j) mit $\underline{i} \leq i < \overline{i}, \underline{j} \leq j \leq \overline{j}$ einen Eintrag verschieden von null, wenn die Gitterpunkte $\{i, j\}$ und $\{i + 1, j\}$ mit einer vertikalen Kante verbunden sind und null sonst. Der eineindeutige Basis-Eckpunkt nach Definition 5.2 gibt die Länge der Kante in der Indexbasis an, die anderen Punkte auf der Kante haben negative Einträge mit dem Betrag des vertikalen Abstandes zum Basis-Eckpunkt der Kante. Analoges gilt für `hEdges`.

- In `tiles` wird jedem Gitterpunkt (i, j) gemäß Definition 5.4 ein festes Element $T \in \mathcal{T}$ zugeordnet. Handelt es sich dabei um den eineindeutigen Basis-Eckpunkt nach Definition 5.2, wird an der Stelle `tiles`$(i, j, 1)$ die horizontale Elementgröße, `tiles`$(i, j, 2)$ die vertikale Elementgröße in der Indexbasis und an `tiles`$(i, j, 3)$ das Element-Level $\ell(T)$ eingetragen. Andernfalls wird der horizontale bzw. vertikale Abstand zum Basis-Eckpunkt mit (-1) multipliziert in `tiles`$(i, j, 1)$ bzw. `tiles`$(i, j, 2)$ eingetragen. In `tiles`$(i, j, 3)$ wird schlicht der Wert null eingetragen.

Daraus lässt sich eine einfache Fallunterscheidung treffen: Ist der eingetragene Wert $\alpha := \texttt{vEdges}(i,j)$ eines Punktes (i,j) positiv, handelt es sich um einen Basiseckpunkt. Ist er negativ, hat die Basiseckpunkt die Koordinaten $(i, j+\alpha)$. Analoges gilt für `hEdges`. Für die Einträge in `tiles` hat der relative Abstand zwischen Gitterpunkt und Basiseckpunkt eines Elementes T jedoch zwei Komponenten $(\alpha_1, \alpha_2) := \texttt{tiles}(i,j,[1,2])$. Der Basiseckpunkt ist dann bei den Koordinaten $(i+\alpha_1, j+\alpha_2)$ zu finden.

Definition 5.4 (EINTRÄGE DER T-GITTER-MATRIZEN). *Für gegebenen Gitterpunkt $P = (i,j)$ eines T-Gitters $\mathcal{T}$ ist die*

- `vertices` *(Abbildung 5.1b):*

$$\texttt{vertices}(i,j) := \begin{cases} 1 & \text{wenn } P \in \mathcal{V}, \\ 0 & \text{sonst.} \end{cases}$$

- `vEdges` *(Abbildung 5.1c):*

$$\texttt{vEdges}(i,j) := \begin{cases} i_2 - i_1 & \text{wenn } \exists\, E = (i_1,i_2) \times \{j\} \in \mathcal{E}^v : P = V_B(E), \\ -(i - i_1) & \text{wenn } \exists\, E = (i_1,i_2) \times \{j\} \in \mathcal{E}^v : P \subset E \backslash \partial E, \\ 0 & \text{sonst.} \end{cases}$$

- `hEdges` *(Abbildung 5.1d):*

$$\texttt{hEdges}(i,j) := \begin{cases} j_2 - j_1 & \text{wenn } \exists\, E = \{i\} \times (j_1,j_2) \in \mathcal{E}^h : P = V_B(E), \\ -(j - j_1) & \text{wenn } \exists\, E = \{i\} \times (j_1,j_2) \in \mathcal{E}^h : P \subset E \backslash \partial E, \\ 0 & \text{sonst.} \end{cases}$$

- `tiles` *(Abbildungen 5.1e, 5.1f und 5.1g):*

$$\begin{pmatrix} \texttt{tiles}(i,j,1) \\ \texttt{tiles}(i,j,2) \end{pmatrix} := \begin{cases} \textbf{\textit{size}}(T) & \text{wenn } \exists\, T \in \mathcal{T} : P = V_B(T), \\ -\begin{pmatrix} i - i_B(T) \\ j - j_B(T) \end{pmatrix} & \text{wenn } \exists\, T \in \mathcal{T} : P \subset E_B(T) \cup T \backslash \partial T, \\ \mathbf{0} & \text{sonst.} \end{cases}$$

$$\texttt{tiles}(i,j,3) := \begin{cases} \ell(T) & \text{wenn } \exists\, T \in \mathcal{T} : P = V_B(T), \\ 0 & \text{sonst.} \end{cases}$$

Die Vorteile dieser Datenstruktur werden erst bei der Implementierung von Algorithmen zur Knoteneinfügung, der Berechnung der Anker oder der visuellen Ausgabe offensichtlich. Die Größe und Position eines Elements an beliebiger Stelle kann durch maximal zwei Lesezugriffe an jeweils einer einzigen Stelle ermittelt werden. Die Anzahl bzw. Position der horizontalen Kanten einer vertikalen Linie (zur Bestimmung des Knotenvektors) sind beispielsweise durch die einfache Bedingung, dass der Eintrag größer als null sein muss, zu ermitteln, wie das nächste Unterkapitel zeigt.

5.1.2. Verfeinerung & Anker

Für eine Übersicht über alle beschriebenen privaten Methoden siehe Listing 5.5.

- `morgensternRefine` startet damit, dass die markierten Elemente in Matrix-Gitterform wie `vertices` (siehe Abbildung 5.1b) gebracht wird. So können zwei Mengen markierter Elemente in Matrixform mittels logischer oder-Verknüpfung vereint werden. Der Ablauf ist im Algorithmus 3.24 festgehalten, nach Satz 3.25 werden AS- und $\mathrm{AD}^+_{\mathbf{p}}$-Eigenschaft erhalten. Folgende Funktionen werden dabei verwendet:
- `markCoarserNeighbours(tile, markedTiles)` bestimmt die gröbere Nachbarschaft des Elements `tile` und markiert alle Elemente in der zurückgegebenen Matrix, die nicht bereits in der Matrix `markedTiles` markiert sind (Listing A.10). Dabei wird die Funktion `dst` verwendet.
- `dst(obj, tileA, tileB)` berechnet den vektorwertigen Abstand zwischen den Mittelpunkten der Elemente `tileA` und `tileB` und gibt diesen zurück (siehe Listing A.11). Dafür wird die Funktion `tileMid` verwendet.
- `tileMid(obj, tile)` retourniert die Koordinaten des Mittelpunkts von `tile` (Listing A.12).
- `simpleRefine(obj, toRefine)` hat als Argument eine Menge von markierten Elementen, die halbiert und in Form eines verfeinerten T-Mesh-Objektes zurückgegeben werden sollen (Listing A.13). Die Elemente werden nacheinander jeweils in einem Schleifendurchlauf verfeinert. Je nachdem, ob das zu verfeinernde Element ganzzahliges oder halbzahliges Level hat, wird es vertikal oder horizontal geteilt. Wenn das Element Kantenlänge eins hat (also die Knotenpunkte der Eckpunkte direkt nebeneinander liegen), wird mit der Funktion `insertVerticalEdge` bzw. `Horizontal` ein Knoten eingefügt vor der Teilung und danach die höheren Indizes in `toRefine` an den neuen Knotenvektor angepasst. Ist der Knoten bereits vorhanden, wird mit der Funktion `addSingleVerticalEdge` bzw. `Horizontal` nur geteilt (Listing A.13).

Die Methode `getAnchors` benötigt die Anker sowie den vertikalen und horizontalen lokalen Indexvektor. Diese werden wiederum aus den jeweiligen vertikalen und horizontalen Indexvektoren berechnet. Die Berechnung der Anker verwendet eine Funktion zur Bestimmung des aktiven Gitterbereichs $\mathrm{AR}_{\mathbf{p}}$. Die korrespondierenden Methoden sind:

- `createAnchors(obj)` berechnet abhängig von den Polynomgraden $\mathbf{p}$ gemäß Definition 3.5 die Anker. Das sind je nach $\mathbf{p}$ alle Ecken (`x`), vertikalen Kanten (`v`), horizontalen Kanten (`h`) oder Elemente (`t`), die im aktiven Bereich $\mathrm{AR}_{\mathbf{p}}$ nach Definition 3.2 liegen (Listing A.14).
- `getHIndexVector(obj, a)` retourniert den horizontalen Indexvektor eines Eckpunktes (einzelner Eintrag in `a`) oder einer Kante (zwei Einträge in `a`: Start- und Endpunkt) nach Definition 3.6 (Listing A.15). Analog für `getHIndexVector(obj, a)` (Listing A.17).

- `getLocalHIndexVector(obj,a)` entnimmt für den Anker (`a` × `b`) gemäß Definition 3.7 Einträge aus dem horizontalen Indexvektor (Listing A.16), der mit der Funktion `getHIndexVector` bestimmt wird . Analoges gilt für die Funktion `getLocalVIndexVector(obj,a)` (Listing A.18).
- `isActiveRegion(obj, x, y)` und `isFrameRegion(obj, x, y)` geben für die gepaarten Vektoren $\mathtt{x} = (x_1, x_2, \ldots, x_\ell)$ und $\mathtt{y} = (y_1, y_2, \ldots, y_\ell)$ einen logischen Vektor zurück, der gemäß Definition 3.2 berechnet, ob die Wertepaare $(x_i, y_i) \in \mathrm{AR}_{\mathbf{p}}$ bzw. $\in \mathrm{FR}_{\mathbf{p}}$ sind. Die Funktionen sind in Listings A.19 bzw. A.20 implementiert.

Listing 5.5 (TMESH.M: VERFEINERUNG & ANKER).

```
34 methods (Access = private)
35   % refinement
36   function [obj, numRefined] = morgensternRefine(obj, initialTiles)
37
38   function erg = markCoarserNeighbours(obj, tile, markedTiles)
39
40   function erg = dst(obj, tileA, tileB)
41
42   function [xErg, yErg] = tileMid(obj, tile)
43
44   function obj = simpleRefine(obj, toRefine)
45
46   % creation of anchors
47   function [result, type] = createAnchors(obj)
48
49   function result = getHIndexVector(obj, a)
50
51   function result = getLocalHIndexVector(obj, a, b)
52
53   function result = getVIndexVector(obj, a)
54
55   function result = getLocalVIndexVector(obj, a, b)
56
57   function result = isActiveRegion(obj, x, y)
58
59   function result = isFrameRegion(obj, x, y)
```

5.1.3. T-Gitter-Manipulation

Alle Funktionen zur Erweiterung des T-Gitters und des Knotenvektors sind nach Eckpunkten, Kanten und Elementen aufgelistet in Listing 5.6.

Zum Einfügen von Knoten in die globalen Knotenvektoren gibt es die folgenden Funktionen:

- `insertKnotX(obj, x)` fügt einen Knotenpunkt zwischen (`x` − 1) und `x` im Knotenvektor `knotsXGlobal` ein und erweitert `vertices`, `vEdges`, `hEdges` und `tiles`

um eine Spalte. Dabei werden an dem Punkt vorhandene horizontale Kanten verlängert und betroffene Elemente vergrößert, sowie deren betroffene Einträge mittels `refreshHEdgeInfo` bzw. `refreshTileInfo` korrigiert (Listing A.21). Analog dazu ist die Funktion `insertKnotY(obj,y)` (Listing A.22) aufgebaut.

- `assureOpenKnotVectors(obj)` erhöht, wenn notwendig, die Multiplizität der Start- und Endknoten 0 und 1 von `knotsXGlobal` bzw. `knotsYGlobal` auf $p_1 + 1$ bzw. $p_2 + 1$ mit den im vorherigen Punkt genannten Funktionen (Listing A.23).

Die Manipulation der Eckpunkte erfolgt durch die folgenden Funktionen:

- `addVertex(obj, x, y)` fügt an der Stelle (x, y) einen Eckpunkt ein und teilt die zugrundeliegende Kante (Listing A.24).
- `isVertex(obj, x, y)` retourniert den Eintrag `vertices(x,y)`, der angibt, ob ein Eckpunkt vorhanden ist (Listing A.25).

Für die Veränderung von Kanten existieren die folgenden Funktionen:

- `splitEdge(obj, x, y)` verkürzt die horizontale/ vertikale Kante am Punkt (x, y), sodass sie an diesem endet und fügt anschließend eine neue Kante am freigewordenen Platz hinzu (Listing A.26).
- `isOnEdge(obj, x, y)` gibt das logische `oder` der beiden Funktionen `isOnVEdge` und `isOnHEdge` zurück (Listing A.27).
- `insertVerticalEdge(obj, xIndex, yEdge)` erweitert zuerst den Knotenvektor in x-Richtung am Punkt `xIndex` mit der Funktion `insertKnotX` und führt anschließend `addSingleVerticalEdge` aus, um das Element zwischen den Gitterpunkten (`xIndex`, `yEdge`[1]) und (`xIndex`, `yEdge`[2]) zu halbieren (Listing A.28). Analog dazu die Funktion `insertHorizontalEdge(obj, xEdge, yIndex)` (Listiing A.33).
- `addSingleVerticalEdge(obj, xIndex, yEdge)` fügt an den beiden Gitterpunkten (`xIndex`, `yEdge`[1]) und (`xIndex`, `yEdge`[2]), sofern nicht vorhanden, Eckpunkte ein und verbindet diese mit einer Kante. Dabei werden bis zu zwei horizontale Kanten geteilt und ein Element, welche mit den dazugehörigen `refreshInfo`-Funktionen aktualisiert werden müssen (Listing A.29). Analog dazu die Funktion `addSingleHorizontalEdge(obj, xEdge, yIndex)` (Listing A.34).
- `refreshVEdgeInfo(obj, x, y)` trägt entlang der vertikalen, durch den Basis-Eckpunkt definierten Kante die Werte gemäß Speicherung von `vEdges` (Definition 5.4) ein (Listing A.30). Die Einträge aller anderen Gitterpunkte bleiben gleich, da es sich an jedem Punkt um relative Angaben handelt. Analog für die Funktion `resfreshHEdgeInfo(obj, x, y)` (Listing A.35).
- `findVEdgeIndex(obj, x, y)` gibt die Koordinaten des Basis-Eckpunktes der Kante, die (`x`, `y`) schneidet, zurück. Wenn es sich bei (`x`, `y`) um einen Basis-Eckpunkt

handelt, so ist dieser damit gefunden. Andernfalls wird der in `vEdges(x,y)` eingetragene (negative) Wert zu `y` addiert, der Basis-Eckpunkt hat dann die Koordinaten $(\mathtt{x}, \mathtt{y} + \mathtt{vEdges}(\mathtt{x}, \mathtt{y}))$ (Listing A.31). Analog dazu `findHEdgeIndex(obj, x, y)` (Listing A.36).

- `isOnVEdge(obj, x, y)` gibt zurück, ob am gegebenen Punkt $(\mathtt{x}, \mathtt{y})$ in `vEdges` eine Zahl verschieden von null eingetragen ist (Listing A.32). An Kantenverläufen sind ausschließlich positive oder negative Zahlen eingetragen. Analog dazu ist die Funktion `isOnHEdge(obj, x, y)` (Listing A.37) definiert.

Die Funktionen bezüglich der Gitter-Elemente sind:

- `findTileIndex(obj, x, y)`, welches auf vektorieller Basis die gleiche Funktionsweise wie `findVEdgeIndex` hat (Listing A.38).
- `refreshTileInfo(obj, x, y)` trägt für alle Punkte, die mit dem Element $T = \{T \in \mathcal{T} : \mathrm{V}_B(T) = (\mathtt{x}, \mathtt{y})$ des Basis-Eckpunkts V_B in der Matrix `tiles` verknüpft sind, die Werte gemäß Speicherung von `tiles` (Definition 5.4) ein. Das sind genau jene Punkte im Inneren des Elementes $\{P = (i, j) : P \in T \backslash \partial T\}$ und die Basis-Kantenpunkte $\mathcal{B}(T)$) (Listing A.39).

Listing 5.6 (TMesh.m: T-Gitter-Manipulation).

```
61  % mesh-manipulation: knotvectors
62  function obj = insertKnotX(obj, x)
63
64  function obj = insertKnotY(obj, y)
65
66  function obj = assureOpenKnotVectors(obj)
67
68  % mesh-manipulation: vertices
69  function obj = addVertex(obj, x, y)
70
71  function result = isVertex(obj, x, y)
72
73  % mesh-manipulation: general edges
74  function obj = splitEdge(obj, x, y)
75
76  function result = isOnEdge(obj, x, y)
77
78  % mesh-manipulation: vertical edges
79  function obj = insertVerticalEdge(obj, xIndex, yEdge)
80
81  function obj = addSingleVerticalEdge(obj, x, yEdge)
82
83  function obj = refreshVEdgeInfo(obj, x, y)
84
85  function [xResultIndex, yResultIndex] = findVEdgeIndex(obj, x, y)
86
87  function result = isOnVEdge(obj, x, y)
```

```
88
89    % mesh-manipulation: horizontal edges
90    function obj = insertHorizontalEdge(obj, xEdge, yIndex)
91
92    function obj = addSingleHorizontalEdge(obj, xEdge, y)
93
94    function obj = refreshHEdgeInfo(obj, x, y)
95
96    function [xResultIndex, yResultIndex] = findHEdgeIndex(obj, x, y)
97
98    function result = isOnHEdge(obj, x, y)
99
100   % mesh-manipulation: tiles
101   function [xResultIndex, yResultIndex] = findTileIndex(obj, x, y)
102
103   function obj = refreshTileInfo(obj, x, y)
104 end
```

5.2. Implementierung des adaptiven Algorithmus

Der adaptive IGAFEM-Algorithmus wurde in der Datei `IGAFEM.m` implementiert, die Auswertung der B-Splines wurde aus Gründen der Performanz in der Sprache C in `evalBspline.c` implementiert und mittels der mex-Bibliothek eingebunden. Ein Überblick über die Funktionen in `IGAFEM.m` ist in Listing 5.7 zu finden.

- `IGAFEM(mesh, f, doerflerTheta)` entspricht einem einzigen Durchlauf des Algorithmus 4.4 ohne die Schritte (iii) und (vi), also ohne Abbruchbedingung und Wiederholung. Argumente sind das anfängliche T-Gitter $\mathcal{T}$(`mesh`), die Funktion $f(x, y)$ (`f`) und der Verfeinerungsparameter θ (`doerflerTheta`). Zurückgegeben werden das verfeinerte T-Gitter $\mathcal{T}'$ (`mesh`), der Fehlerschätzer η (`error`) und der Energiefehler E (`energy`) der Lösung. Der Energiefehler wird im nächsten Kapitel 6 erklärt und in Definition 6.1 definiert. Siehe Listing B.40 für dessen Implementierung.

- `evalULaplace(X, Y, anchorPoints, xKnots, yKnots, p, q)` wertet die zweiten partiellen Ableitungen der Gitterfunktion $u_\ell(x, y)$ an den gegebenen Punkten $(\texttt{X}[i], \texttt{Y}[i])_{i=1,2,\dots,n}$ aus, also $\partial_x^2 u_\ell(x, y)$ und $\partial_y^2 u_\ell(x, y)$ und gibt diese zurück. Die Argumente `anchorPoints`, `xKnots` und `yKnots` sind zeilenweise zusammengehörende Matrizen mit allen Ankerpunkten (dreidimensional) und den dazugehörigen lokalen Knotenvektoren. Zum Anker $A_j \in \mathcal{A}(\mathcal{T})$ mit Koordinaten `anchorPoints`$[j, [1, 2]]$ gehören `knotsX`$[j, :]$ und `knotsY`$[j, :]$. Die Argumente `p` und `q` sind die Polynomgrade in x- und y-Richtung. Siehe Listing B.41.

- `evalBDerive(knots, p, x)` wertet die erste partielle Ableitung des einzelnen B-Splines $\partial_\zeta B_{0,\texttt{p}}[\texttt{knots}](\zeta)$ an den Stellen $(\texttt{x}[i])_{i=1,2,\dots,n}$ aus und gibt diese in einer Matrix mit den gleichen Dimensionen wie `x` zurück (Listing B.42).

- `evalBLaplace(knots, p, x)` ist ident zu `evalBDerive(knots, p, x)`, nur dass die zweiten partiellen Ableitungen anstelle der ersten zurückgegeben werden (Listing B.43).
- `gaussLegendre(a, b, degree)` berechnet die $n+1$ Gewichte und Auswertungspunkte der Gauß-Legendre-Quadratur vom Grad `degree` $=: n$ im Intervall $[\mathtt{a}, \mathtt{b}]$ zurück. Für Grad n hat die Quadratur den maximalen Exaktheitsgrad $2n+1$, also Polynome bis zum Grad $2n+1$ werden exakt integriert. Siehe [10] für Details und Konstruktion, sowie Listing B.44 für dessen Implementierung.
- `evalBSpline(xi, p, x, i)` aus der Datei `evalBSpline.c` wertet die B-Spline-Funktion $B_{\mathtt{i},\mathtt{p}}(\zeta)$ des `p`-offenen Knotenvektor `xi` an den Stellen $(\mathtt{x}[\ell])$ für alle $\ell = 1, 2, \ldots, n$ basierend auf dem Neville-Verfahren aus und gibt das Ergebnis zurück. Siehe Algorithmus 5.8 und Listing C.45 für Details und Implementierung.

Listing 5.7.

```
1  function [mesh, error, energy] = IGAFEM(mesh, f, doerflerTheta)
2
3  function [laplaceX, laplaceY] = evalULaplace...
4      (X, Y, anchorPoints, xKnots, yKnots, p, q)
5
6  function res = evalBDerive(knots,p,x)
7
8  function res = evalBLaplace(knots,p,x)
9
10 % Gauss-Legendre
11 function [weights, nodes] = gaussLegendre(a, b, degree)
12
13 function res = polynomProduct(s,t)
14
15 function res = innerProduct(s, t, a, b)
16
17 function res = polyInt(coeff, a, b)
```

Algorithmus 5.8 (B-SPLINE-AUSWERTUNG).

Input: *Knotenvektor Ξ, Polynomgrad p, Auswertungspunkte x, B-Spline-Index i Für jeden Punkt x_ℓ in x mache:*

> ***Schleife:*** *Für $j = 0, 1, \ldots, p$ mache:*
>
> $\mathtt{val}[j] := 1$, *wenn* $\mathtt{xi}[\mathtt{i}] \leq x_\ell < \mathtt{xi}[\mathtt{i}+1]$,
>
> $\mathtt{val}[j] := 0$ *sonst.*
>
> ***Schleife:*** *Für $j = 1, 2, \ldots, p$ mache:*
>
> > ***Schleife:*** *Für $k = 1, 2, \ldots, p - j$ mache:*

$\texttt{val}[k] := B_{k,\texttt{i}+j}(x_\ell)$ *nach Definition 2.2 mit* $B_{k,\texttt{i}+j-1}(x_\ell) := \texttt{val}[k]$ *und* $B_{k+1,\texttt{i}+j-1}(x_\ell) := \texttt{val}[k+1]$).

$\texttt{result}[\ell] := \texttt{val}[0]$

Output: *Funktionswerte* $\texttt{result}[\ell] = B_{\texttt{i},\texttt{p}}(x_\ell)$

6. Numerische Beispiele

In diesem Kapitel werden beispielhafte Rechnungen für eine glatte Funktion und eine nicht-glatte Funktion $f(x, y)$ für die rechte Seite der Poisson-Gleichung nach Definition 4.1 präsentiert. Zuerst müssen noch die verwendeten Begriffe eingeführt werden.

Definition 6.1 (Energiefehler). *Der Energiefehler δE ist für gegebene Galerkin-Approximation $u_h \in U_h \subset H_0^1$ auf dem T-Gitter $\mathcal{T}_h$ eines gegebenen Poisson-Problems mit exakter Lösung $u(x, y)$ gegeben durch*

$$\delta E := ||\nabla(u - u_h)||_{L^2} = \sqrt{||\nabla u||_{L^2}^2 - ||\nabla u_h||_{L^2}^2}$$

und kann mit gegebener Referenzenergie E_{ref} (berechnet mittels $\mathcal{T} = \mathcal{T}_{u[k]} für k > \max_{T \in \mathcal{T}_h} \ell(T)$) angenähert werden durch

$$||\nabla u||_{L^2}^2 - ||\nabla u_h||_{L^2}^2 \approx \underbrace{||\nabla u_{ref}||_{L^2}^2}_{=:E_{ref}} - ||\nabla u_h||_{L^2}^2.$$

Definition 6.2 (Maximaler Punktfehler). *Für gegebene Galerkin-Lösung $u_h \in U_h \subset H_0^1$ auf dem T-Gitter $\mathcal{T}_h$ eines gegebenen Poisson-Problems mit exakter Lösung $u(x, y)$ ist der maximale Punktfehler gegeben durch*

$$\|u - u_h\|_h := \max_{(x,y) \in \mathcal{V}_h} |u(x, y) - u_h(x, y)|,$$

Definition 6.3 (Experimentelle Konvergenzrate). *Für N Datenpunkte $P_i = (x_i, y_i)_{i=1,2,\ldots,N}$ ist die Rate $r > 0$ als jener Wert definiert, der die Summe der logarithmischen Abstandsquadrate*

$$\sum_{i=1}^{N} \| \log(y_i) - \log(g_{a,r}(x_i)) \|^2$$

minimiert, mit

$$g_{a,r}(x) = (ax)^r,$$

für ebenfalls gesuchtes $a \in \mathbb{R}^+ \backslash \{0\}$.

In den folgenden Beispielen wurde die experimentelle Konvergenzrate nach Definition 6.3 mittels `fit`-Funktion von Matlab mit der Methode `NonlinearLeastSquares` und dem Algorithmus `Trust-Region` verwendet.

6.1. Glattes Beispiel: Sinusfunktion

Vorgegeben ist das Poisson-Problem nach Definition 4.1 mit der Funktion

$$f(x, y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y)$$

und der zugehörigen exakten Lösung

$$u(x, y) = \sin(2\pi x) \sin(2\pi y).$$

Die Abbildungen 6.1 ($\theta = 0.5$) und 6.3 ($\theta = 1$) zeigen jeweils Fehlerschätzer η und Energiefehler δE für verschiedene Polynomgrade $p \in \{2, 3, 4, 5\}$. Für gleiches p sind η und δE jeweils parallel. Das unerwartete Verhalten in den letzten Datenpunkten des Energiefehlers δE für $p = 5, \theta = 0.5$ ist auf die begrenzte Genauigkeit von `double` zurückzuführen. Die absoluten Energiewerte $E = \mathbf{x}^T \mathbf{A} \mathbf{x}$ (mit Lösungsvektor $\mathbf{x}$ und Galerkin-Matrix $\mathbf{A}$) sind in Tabelle 6.1 zu sehen. Die Galerkin-Matrix $\mathbf{A}$ hat allerdings bis zu ungefähr $N = (4 \cdot 10^3) \times (4 \cdot 10^3) = 1.6 \cdot 10^7$ Einträge, über die summiert wird. Der Energiefehler δE konvergiert theoretisch gegen null mit $\delta E_i \leq \delta E_{i-1}$. Dass diese Bedingung aufgrund von Quadraturfehlern und Rechenungenauigkeit nicht erfüllt wird, ist genauer in Abbildung 6.2 und in der dazugehörigen Tabelle 6.1 zu sehen.

In den Abbildungen 6.4a ($\theta = 0.5$) und 6.4b ($\theta = 1$) ist der maximale Punktfehler nach Definition 6.2 für verschiedene Polynomgrade $p \in \{2, 3, 4, 5\}$ zu sehen.

6.2. Glattes Beispiel: Konstante Funktion

Vorgegeben ist das Poisson-Problem nach Definition 4.1 mit der Funktion

$$f(x, y) = 1.$$

Die erste Abbildungsreihe vergleicht den Fehlerschätzer η verschiedener Polynomgrade $p = \{2, 3, 4, 5\}$ je einmal für adaptive Verfeinerung mit $\theta = 0.5$ und für uniforme Verfeinerung mit $\theta = 1$. Der Satz mit uniformer Verfeinerung verhält sich dabei, wie es zu erwarten ist: η und δE sind für gleiches p jeweils parallel (Abbildung 6.6). Bei adaptiver Verfeinerung verlaufen η und δE anfangs parallel, aber über den Wert $\delta E \approx 10^{-7}$ hinaus wird der Energiefehler nicht mehr kleiner. Mit höherem Polynomgrad wird der Grenzwert früher erreicht, für $p = 2$ (Abbildung 6.5a) nicht innerhalb der Experiments, für $p = 5$ (Abbildung 6.5d) bei der halben logarithmischen Messreihe und $p = 3, 4$ (Abbildung 6.5b, 6.5c) dazwischen.

6.3. Nicht-glattes Beispiel

Vorgegeben ist das Poisson-Problem nach Definition 4.1 mit der Indikatorfunktion

$$f(x, y) = \begin{cases} 1 & \text{auf} \quad \Omega_1 = [0, 0.5]^2, \\ 0 & \text{sonst} \end{cases}$$

Diese Abbildungsserie vergleicht für die Polynomgrade $p = \{2, 3, 4, 5\}$ bei adaptiver bzw. uniformer Verfeinerung mit $\theta = 0.5$ bzw. $\theta = 0.5$ (Abbildung 6.7 bzw. 6.8) den Fehlerschätzer η und den Energiefehler δE.

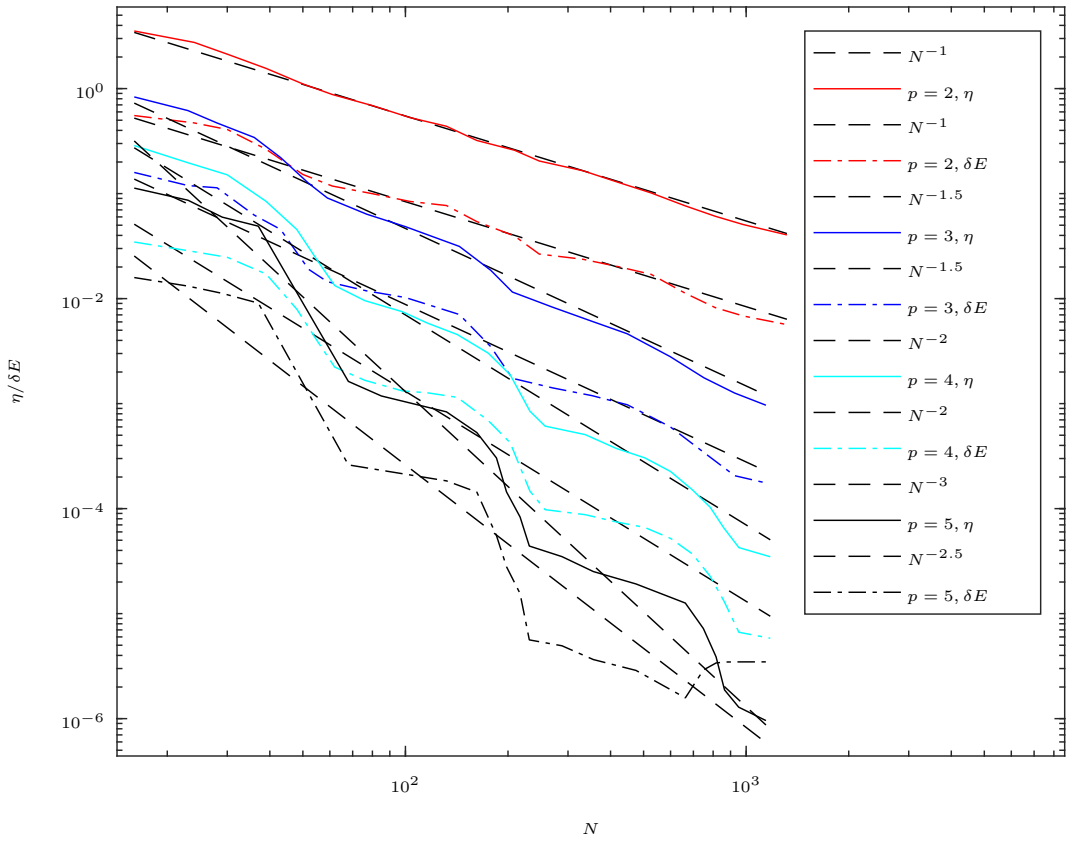


Abbildung 6.1.: Numerische Beispiele zu Abschnitt 6.1: Fehlerschätzer η und Energiefehler δE für adaptive Verfeinerung ($\theta = 0.5$) im Vergleich verschiedener Polynomgrade $p \in \{2, 3, 4, 5\}$ für glatte Funktion $f(x, y)$. Die Punkte des Energiefehlers δE für $p = 5$ sollten monoton fallend und parallel zum Fehlerschätzer η sein, sind jedoch aufgrund der Rechengenauigkeit fehlerhaft, siehe Abbildung 6.2 und Tabelle 6.1 für Details dazu.

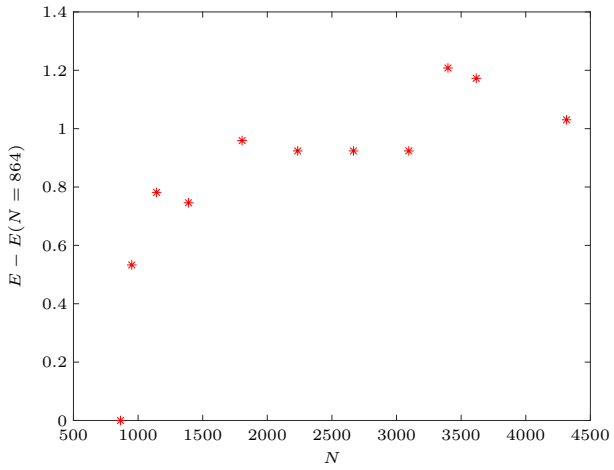


Abbildung 6.2.: Absoluter Energiewert (kleinster Energiewert auf Nullpunkt verschoben) für Abschnitt 6.1, für $\theta = 0.5$ und $p = 5$. Energiewerte E mit größerem N sollten strikt größer sein, sind es aber teilweise aufgrund der begrenzten Rechengenauigkeit nicht. Siehe Tabelle 6.1 für die absoluten Werte.

N	E
864	19.739208802179373
951	19.739208802179427
1143	19.739208802179451
1390	19.739208802179448
1805	19.739208802179469
2235	19.739208802179466
2667	19.739208802179466
3095	19.739208802179466
3398	19.739208802179494
3617	19.739208802179490
4316	19.739208802179476

Tabelle 6.1.: Absolute Energiewerte zu Abschnitt 6.1 für $\theta = 0.5$ und $p = 5$.

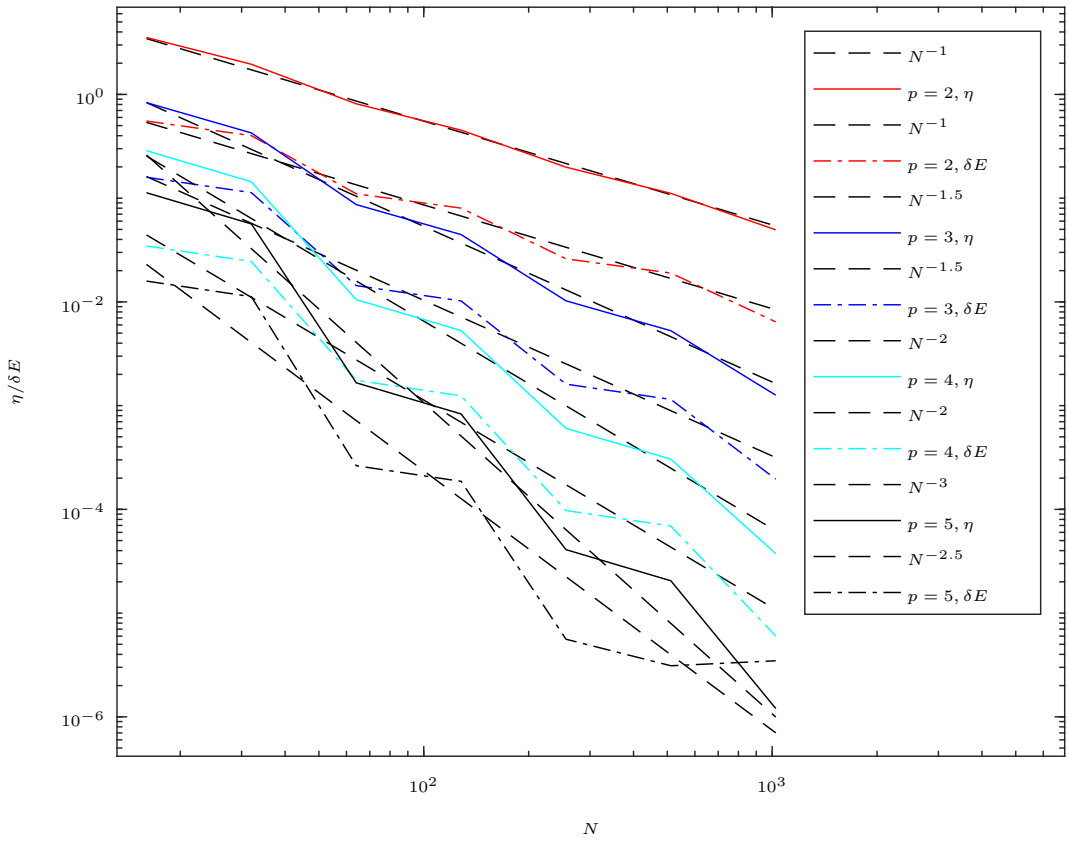
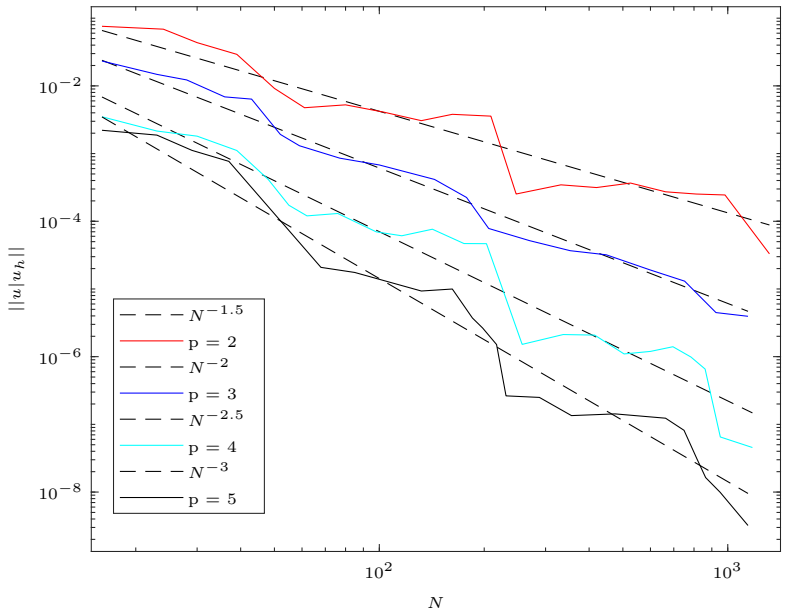
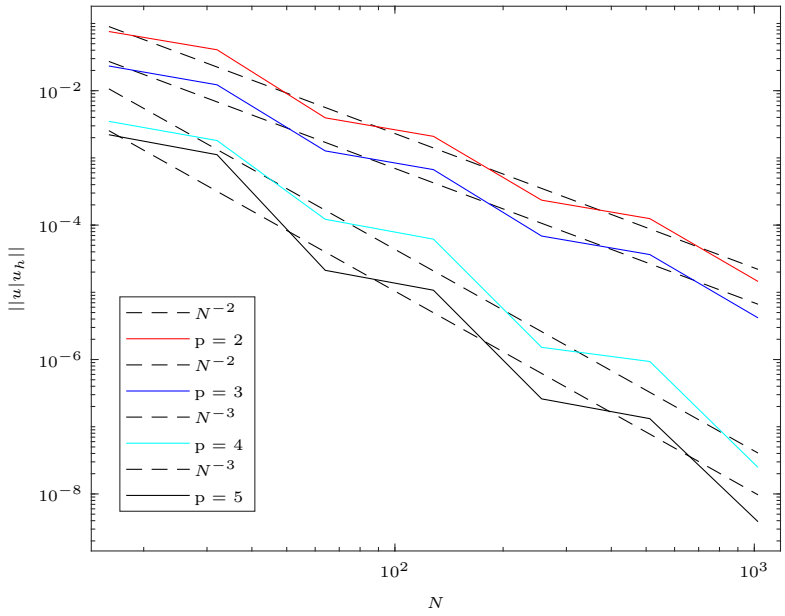


Abbildung 6.3.: Numerische Beispiele zu Abschnitt 6.1: Fehlerschätzer η und Energiefehler δE für uniforme Verfeinerung im Vergleich verschiedener Polynomgrade $p \in \{2, 3, 4, 5\}$ für glatte Funktion $f(x, y)$.

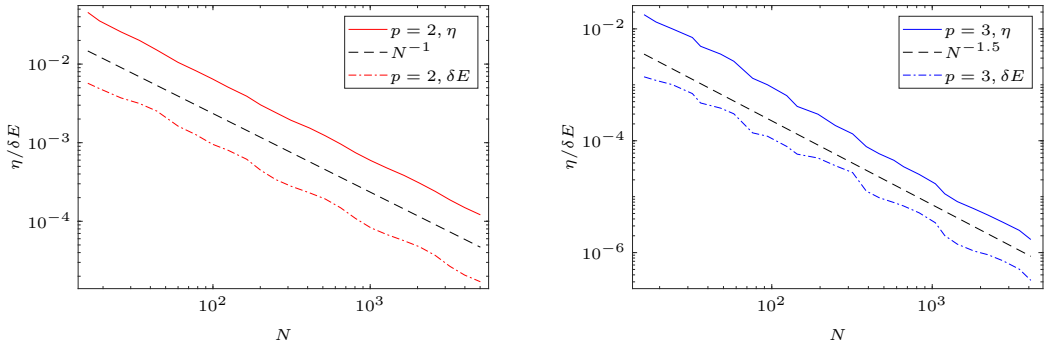


(a) Adaptiv, $\theta = 0.5$

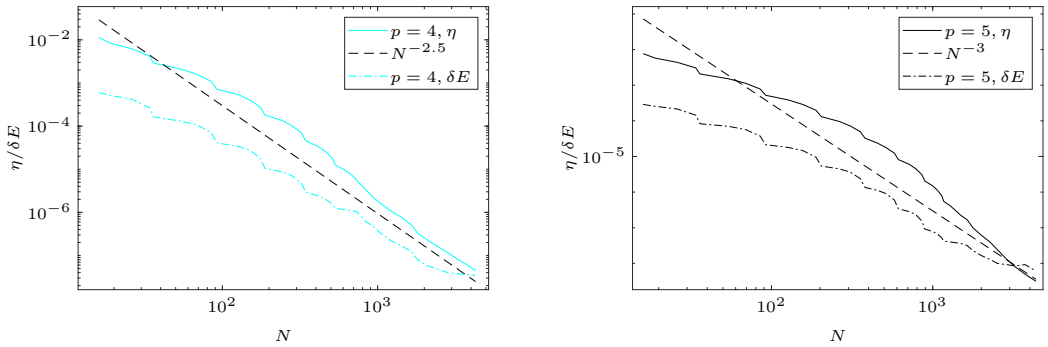


(b) Uniform, $\theta = 1$

Abbildung 6.4.: Numerische Beispiele zu Abschnitt 6.1: Maximaler Punktfehler für (a) adaptive bzw. (b) uniforme Verfeinerung ($\theta = 0.5$ bzw. $\theta = 1$) im Vergleich verschiedener Polynomgrade $p \in \{2, 3, 4, 5\}$ für glatte Funktion $f(x, y)$.



(a) $p = 2$ (b) $p = 3$



(c) $p = 4$ (d) $p = 5$

Abbildung 6.5.: Numerische Beispiele zu Abschnitt 6.2: Fehlerschätzer η und Energiefehler E_{error} für adaptive Verfeinerung ($\theta = 0.5$) für verschiedene Polynomgrade $p = \{2, 3, 4, 5\}$ und konstante $f(x, y)$.

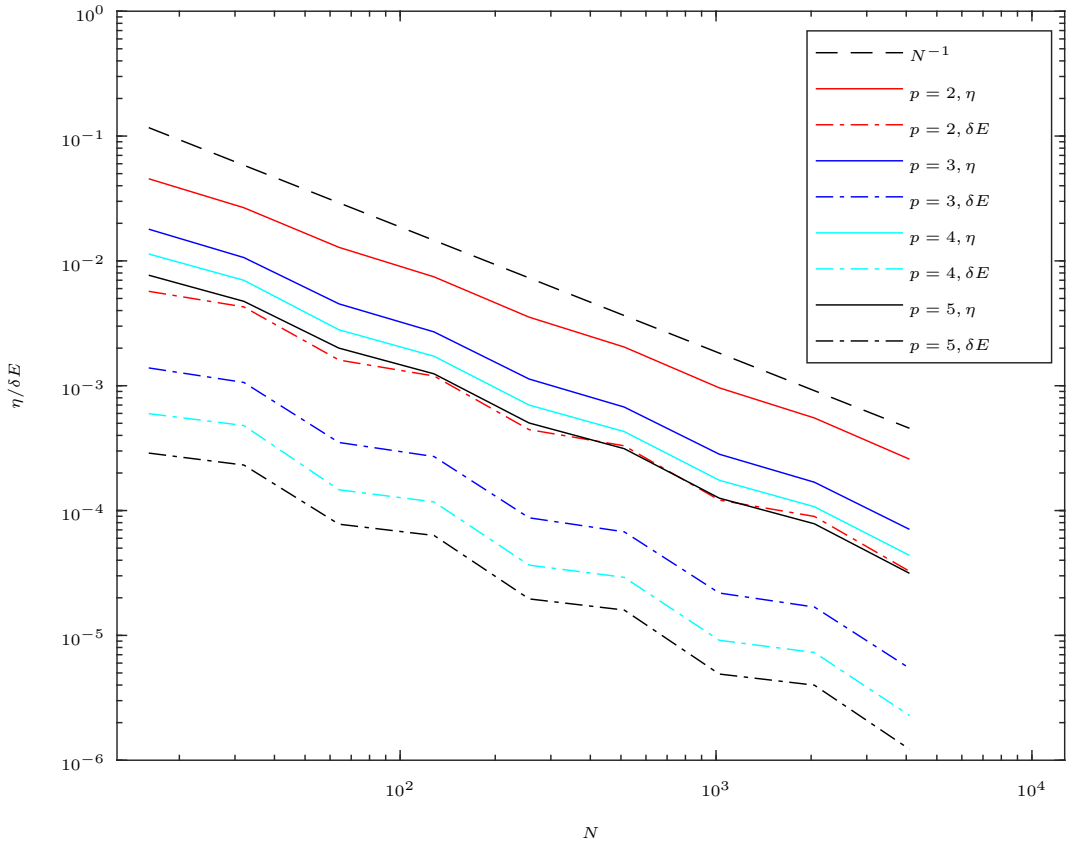
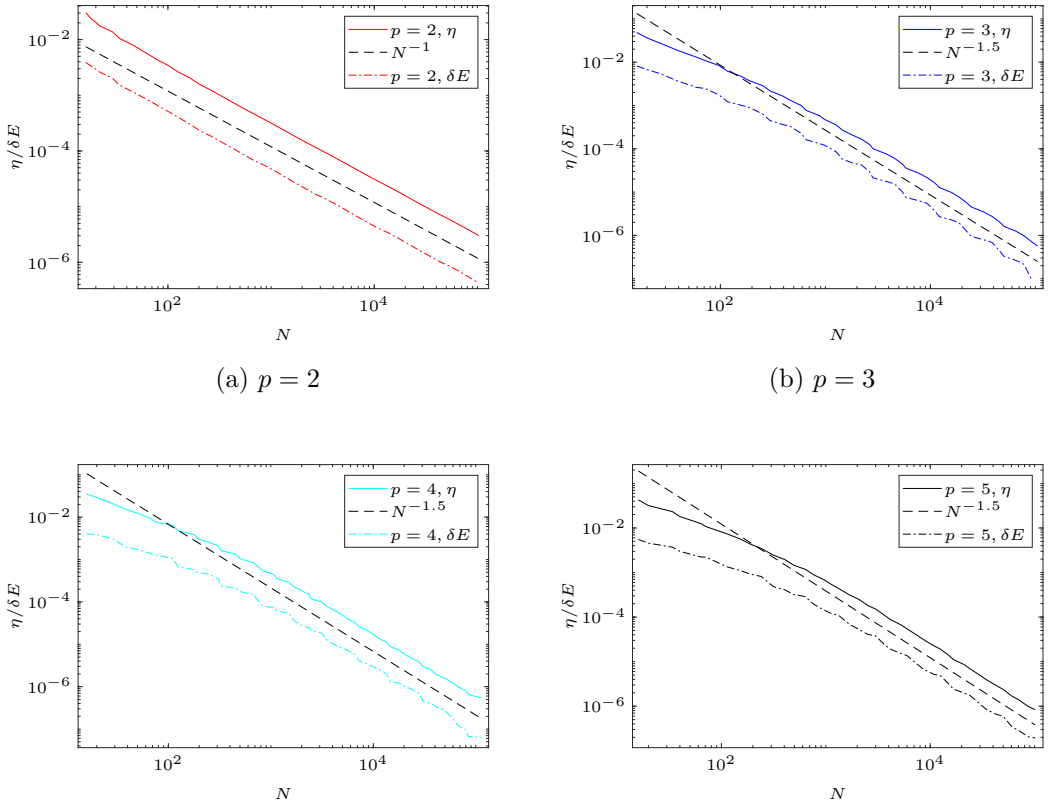


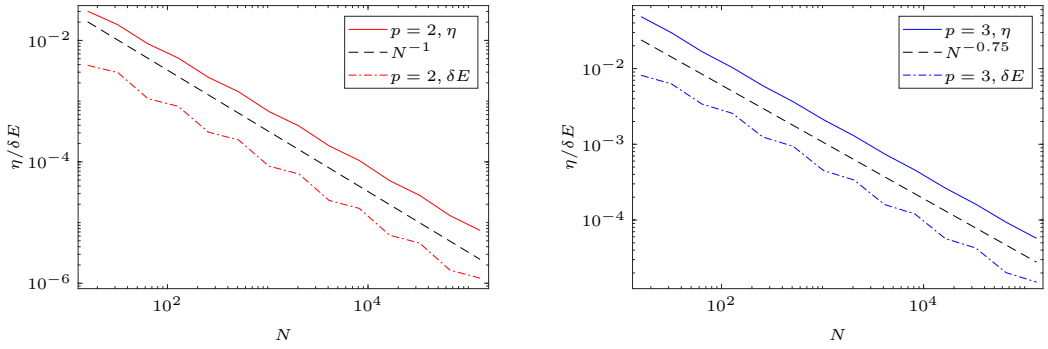
Abbildung 6.6.: Numerische Beispiele zu Abschnitt 6.2: Fehlerschätzer η für uniforme Verfeinerung im Vergleich verschiedener Polynomgrade $p \in \{2, 3, 4, 5\}$ für konstante Funktion $f(x, y)$.



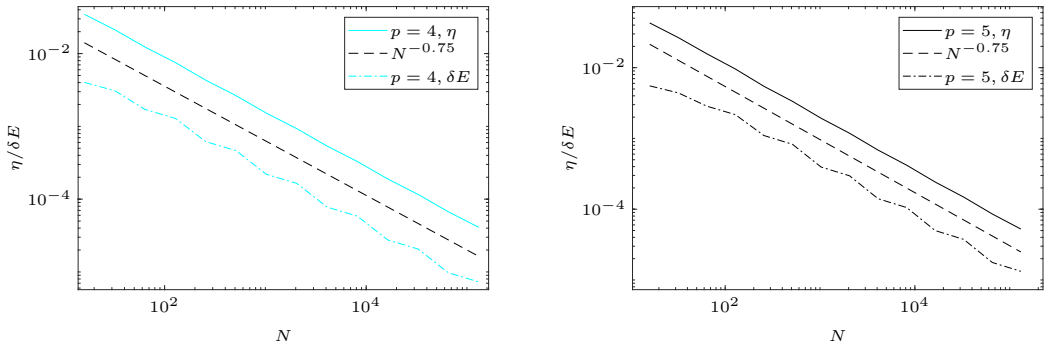
(a) $p = 2$ (b) $p = 3$

(c) $p = 4$ (d) $p = 5$

Abbildung 6.7.: Numerische Beispiele zu Abschnitt 6.3: Fehlerschätzer η und Energiefehler δE für adaptive Verfeinerung ($\theta = 0.5$) für verschiedene Polynomgrade und nicht-glatter Indikatorfunktion $f(x, y)$.



(a) $p = 2$ (b) $p = 3$



(c) $p = 4$ (d) $p = 5$

Abbildung 6.8.: Numerische Beispiele zu Abschnitt 6.3: Fehlerschätzer η und Energiefehler δE für uniforme Verfeinerung für verschiedene Polynomgrade und nicht-glatter Indikatorfunktion $f(x, y)$.

Literaturverzeichnis

[1] Beirão da Veiga, L., A. Buffa, G. Sangalli und R. Vázquez: *Analysis-suitable T-splines of arbitrary degree: definition, linear independence and approximation properties*. Math. Models Methods Appl. Sci., 23(11):1979–2003, 2013.

[2] Beirão da Veiga, L., A. Buffa, G. Sangalli und R. Vázquez: *Mathematical analysis of variational isogeometric methods*. Acta Numer., 23:157–287, 2014.

[3] Binev, P., W. Dahmen und R. DeVore: *Adaptive finite element methods with convergence rates*. Numer. Math., 97(2):219–268, 2004.

[4] Bressan, A.: *Some properties of LR-splines*. Comput. Aided Geom. Design, 30(8):778–794, 2013.

[5] Buffa, A., D. Cho und G. Sangalli: *Linear independence of the T-spline blending functions associated with some particular T-meshes*. Comput. Methods Appl. Mech. Engrg., 199(23-24):1437–1445, 2010.

[6] Carstensen, C., M. Feischl, M. Page und D. Praetorius: *Axioms of adaptivity*. Comput. Math. Appl., 67(6):1195–1253, 2014.

[7] Cottrell, J. A., T. J. R. Hughes und Y. Bazilevs: *Isogeometric analysis*. John Wiley & Sons, Ltd., Chichester, 2009. Toward integration of CAD and FEA.

[8] Hughes, T. J. R., J. A. Cottrell und Y. Bazilevs: *Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement*. Comput. Methods Appl. Mech. Engrg., 194(39-41):4135–4195, 2005.

[9] Morgenstern, P.: *Mesh Refinement Strategies for the Adaptive Isogeometric Method*. Doktorarbeit, Institut für Numerische Simulation, Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[10] Praetorius, D.: *Adaptive Finite-Elemente-Methode, Vorlesungsskriptum*. Institut für Analysis und Scientific Computing, TU Wien, 2015.

[11] Schumaker, L. L.: *Spline functions: basic theory*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, Dritte Aufl., 2007.

A. Anhang: Listings

A.1. TMesh.m

A.1.1. Konstruktor

Listing A.1 (TMesh.m: Konstruktor).

```
19 function obj = TMesh(initLevel, deg)
20     if(mod(initLevel,1) == 0.5)
21         raiseHalveLevel = true;
22         initLevel = initLevel -0.5;
23     else
24         raiseHalveLevel = false;
25     end
26     assert(initLevel >= 0 && mod(initLevel,1) == 0);
27     initEl = 2^(initLevel);
28     extGrid = initEl + 2*deg + 1;
29
30     % global open knotvectors
31     openVec = [zeros(1,deg), linspace(0,1,initEl+1), ones(1,deg)];
32     obj.knotsXGlobal = openVec;
33     obj.knotsYGlobal = openVec;
34     obj.initialTileSize = 1/initEl;
35
36     % vertices
37     obj.vertices = ones(extGrid, extGrid);
38
39     % edges
40     obj.vEdges = [ ones(extGrid, extGrid-1), -ones(extGrid, 1) ];
41     obj.hEdges = [ ones(extGrid-1, extGrid); -ones(1, extGrid) ];
42
43     % tiles
44     obj.tiles = ones(extGrid, extGrid, 3);
45     obj.tiles(2:extGrid, extGrid, [1 2]) = -1;
46     obj.tiles(extGrid, 2:extGrid, [1 2]) = -1;
47     obj.tiles(1, extGrid, [1 2]) = [0 -1];
48     obj.tiles(extGrid, 1, [1 2]) = [-1 0];
49
50     % tile level, frame region default: -1
51     obj.tiles(:,:,3) = initLevel;
52     obj.tiles(1:(end-1),[1:deg,(end-deg):(end-1)],3) = -1;
53     obj.tiles([1:deg,(end-deg):(end-1)],1:(end-1),3) = -1;
54
55     % polynomial degree
56     obj.degree = [deg,deg];
```

```
57      obj.assureOpenKnotVectors();
58
59      if(raiseHalveLevel == true)
60          obj.refineTiles(obj.getNonzeroActiveTiles());
61      end
62  end
```

A.1.2. Öffentlich: Methoden

Listing A.2 (TMesh.m: getAnchors).

```
64  function [anchors, localX, localY] = getAnchors(obj)
65      [indexAnchors, type] = obj.createAnchors();
66      [xBase, yBase] = find(indexAnchors);
67      anchors = zeros(length(xBase),2);
68      isInitialised = false;
69      for i = 1:length(xBase)
70          x = xBase(i);
71          y = yBase(i);
72          switch type
73              case 't'
74                  a = [x, x + obj.tiles(x,y,1)];
75                  b = [y, y + obj.tiles(x,y,2)];
76              case 'h'
77                  a = [x, x + obj.hEdges(x,y)];
78                  b = y;
79              case 'v'
80                  a = x;
81                  b = [y, y + obj.vEdges(x,y)];
82              case 'x'
83                  a = x;
84                  b = y;
85          end
86          if(isInitialised == false)
87              hLength = length(obj.getLocalHIndexVector(a,b));
88              vLength = length(obj.getLocalVIndexVector(a,b));
89              localX = zeros(length(xBase),hLength);
90              localY = zeros(length(xBase),vLength);
91              isInitialised = true;
92          end
93          xMid = 0.5*sum(obj.knotsXGlobal([a(1),a(end)]));
94          yMid = 0.5*sum(obj.knotsYGlobal([b(1),b(end)]));
95          anchors(i,[1,2]) = [xMid,yMid];
96          localX(i,:) = obj.knotsXGlobal(obj.getLocalHIndexVector(a,b));
97          localY(i,:) = obj.knotsYGlobal(obj.getLocalVIndexVector(a,b));
98      end
99  end
```

Listing A.3 (TMesh.m: getUniqueGlobalKnots).

```
101 function [xGlobal, yGlobal] = getUniqueGlobalKnots(obj)
102     xGlobal = unique(obj.knotsXGlobal);
103     yGlobal = unique(obj.knotsYGlobal);
104 end
```

Listing A.4 (TMesh.m: getVertices).

```
106 function [xValues, yValues] = getVertices(obj)
107     [xIndex, yIndex] = find(obj.vertices);
108     xCoord = obj.knotsXGlobal(xIndex);
109     yCoord = obj.knotsYGlobal(yIndex);
110     xValues = obj.gamma_x(xCoord, yCoord);
111     yValues = obj.gamma_y(xCoord, yCoord);
112 end
```

Listing A.5 (TMesh.m: getMaxTileLevel).

```
114 function maxTileLevel = getMaxTileLevel(obj)
115     maxTileLevel = max(obj.tiles(:,:,3),[], 'all');
116 end
```

Listing A.6 (TMesh.m: getActiveTiles).

```
118 function result = getActiveTiles(obj)
119     [spX, spY] = find(obj.tiles(:,:,1)>0);
120     epX = 0*spX;
121     epY = 0*spY;
122     for i = 1:length(epX)
123         epX(i) = spX(i) + obj.tiles(spX(i),spY(i),1);
124         epY(i) = spY(i) + obj.tiles(spX(i),spY(i),2);
125     end
126     startpointsAR = obj.isActiveRegion(spX, spY);
127     endpointsAR = obj.isActiveRegion(epX, epY);
128     tilesAR = and(startpointsAR, endpointsAR);
129     coordSP = [obj.knotsXGlobal(spX(tilesAR))',...
130         obj.knotsYGlobal(spY(tilesAR))'];
131     coordEP = [obj.knotsXGlobal(epX(tilesAR))',...
132         obj.knotsYGlobal(epY(tilesAR))'];
133     result = [coordSP, coordEP];
134 end
```

Listing A.7 (TMesh.m: getNonzeroActiveTiles).

```
136 function result = getNonzeroActiveTiles(obj)
137     activeTiles = obj.getActiveTiles();
138     nonzero = and(activeTiles(:,1) < activeTiles(:,3),...
139         activeTiles(:,2) < activeTiles(:,4));
140     result = activeTiles(nonzero,:);
141 end
```

Listing A.8 (TMesh.m: refineTiles).

```
143 function [obj, numRefined] = refineTiles(obj, toRefine)
144     indexToRefine = 0*toRefine;
145     for i = 1:size(toRefine,1)
146         xLower = find((obj.knotsXGlobal <= toRefine(i,1)),1,'last');
147         yLower = find((obj.knotsYGlobal <= toRefine(i,2)),1,'last');
148         [indexToRefine(i,1), indexToRefine(i,2)] = ...
149             obj.findTileIndex(xLower, yLower);
150     end
151     [obj, numRefined] = obj.morgensternRefine(indexToRefine);
152 end
```

A.1.3. Privat: Funktionen zur Verfeinerung

Listing A.9 (TMesh.m: (private) morgensternRefine).

```
162 function [obj, numRefined] = morgensternRefine(obj, initialTiles)
163     % index-based
164     markedTiles = 0*obj.tiles(:,:,1);
165 
166     % finding the maximum level
167     max_level = -1;
168     for i = 1:size(initialTiles,1)
169         x = initialTiles(i,1);
170         y = initialTiles(i,2);
171         assert(obj.tiles(x,y,1)>0);
172         level = obj.tiles(x,y,3);
173         if(level > max_level)
174             max_level = level;
175         end
176         markedTiles(x,y) = 1;
177     end
178     newMarked = markedTiles;
179 
180     % marking the n-th coarse neighbourhood
181     for i = 0:(2*max_level)
182         [markedX, markedY] = find(newMarked);
183         newMarked = 0*newMarked;
184 
185         % marking the neighbourhood of a single tile
186         for j = 1:size(markedX,1)
187             tile = [markedX(j),markedY(j)];
188             allMarked = or(newMarked(:,:),markedTiles(:,:));
189             partialSol = obj.markCoarserNeighbours(tile,allMarked);
190             newMarked = or(newMarked, partialSol);
191         end
192         markedTiles = or(markedTiles, newMarked);
193         if(sum(newMarked,'all') == 0)
194             break;
195         end
```

```
196     end
197
198     % actual refinement process
199     [toRefine(:,1), toRefine(:,2)] = find(markedTiles);
200     obj.simpleRefine(toRefine);
201     numRefined = size(toRefine,1);
202 end
```

Listing A.10 (TMesh.m: (private) markCoarserNeighbours).

```
204 function erg = markCoarserNeighbours(obj, tile, markedTiles)
205     if(nargin == 2)
206         markedTiles = 0*obj.tiles(:,:,1);
207     end
208
209     % allocating the result
210     erg = 0*obj.tiles(:,:,1);
211     level = obj.tiles(tile(1),tile(2),3);
212     targetLevel = level - 0.5;
213     assert(level >= 0);
214
215     % defining the maxDst vector
216     p = obj.degree(1);
217     if(mod(level,1) == 0)
218         maxDst = 2^(-level)*[floor(p/2)+0.5,ceil(p/2)+0.5];
219     else
220         assert(mod(level,1) == 0.5)
221         maxDst = 2^(-level-0.5)*[ceil(p/2)+0.5, 2*floor(p/2)+1];
222     end
223
224     % applying the level criteria to all tiles
225     toConsider = and(obj.tiles(:,:,1)>0, ~markedTiles);
226     toConsider = and(obj.tiles(:,:,3) == targetLevel, toConsider);
227     [x, y] = find(toConsider);
228
229     % applying the distance criteria to the remaining tiles
230     allDst = obj.dst(tile, [x,y]);
231     found = find(and(allDst(:,1) <= maxDst(1),...
232         allDst(:,2) <= maxDst(2)));
233     % marking the found tiles in result
234     for i = 1:length(found)
235         index = found(i);
236         erg(x(index),y(index)) = 1;
237     end
238 end
```

Listing A.11 (TMesh.m: (private) dst).

```
240 function erg = dst(obj, tileA, tileB)
241     % vector-valued distance between two tiles
242     erg = zeros(size(tileB,1),2);
```

```
243     [axMid, ayMid] = obj.tileMid(tileA);
244     for i = 1:size(tileB,1)
245         [bxMid, byMid] = obj.tileMid(tileB(i,:));
246         erg(i,[1,2]) = abs([axMid-bxMid, ayMid-byMid]);
247     end
248 end
```

Listing A.12 (TMesh.m: (private) tileMid).

```
250 function [xErg, yErg] = tileMid(obj, tile)
251     % returns the coordinates of a given tile's mid point
252     assert(length(tile) == 2);
253     assert(obj.tiles(tile(1),tile(2),1) > 0);
254     xIndexEnd = tile(1) + obj.tiles(tile(1),tile(2), 1);
255     xErg = 0.5*sum(obj.knotsXGlobal([tile(1), xIndexEnd]));
256     yIndexEnd = tile(2) + obj.tiles(tile(1), tile(2), 2);
257     yErg = 0.5*sum(obj.knotsYGlobal([tile(2), yIndexEnd]));
258 end
```

Listing A.13 (TMesh.m: (private) simpleRefine).

```
260 function obj = simpleRefine(obj, toRefine)
261 % splits every given tile from toRefine
262 for i = 1:size(toRefine,1)
263     xStart = toRefine(i,1);
264     yStart = toRefine(i,2);
265     assert(obj.tiles(xStart,yStart,1)>0);
266     xEnd = xStart + obj.tiles(xStart,yStart,1);
267     yEnd = yStart + obj.tiles(xStart,yStart,2);
268     level = obj.tiles(xStart,yStart,3);
269     if(mod(level,1) == 0)
270         % x-refinement, i. e. add/insert a vertical edge
271         if(xStart == xEnd-1)
272           obj.insertVerticalEdge(xEnd, [yStart, yEnd], false);
273           % update unrefined vertices from old indexing
274           % due to insertation of new edge
275           update = find([zeros(i,1);toRefine((i+1):end,1) >= xEnd]);
276           toRefine(update,1) = toRefine(update,1) + 1;
277         else
278           xCStart = obj.knotsXGlobal(xStart);
279           xCEnd = obj.knotsXGlobal(xEnd);
280           xCMiddle = 0.5*(xCStart+xCEnd);
281           xMiddle = find(obj.knotsXGlobal == xCMiddle);
282           assert(length(xMiddle) == 1);
283           obj.addSingleVerticalEdge(xMiddle, [yStart, yEnd]);
284         end
285     else
286         assert(mod(level,1) == 0.5);
287         % y-refinement, i. e. add/insert a horizontal edge
288         if(yStart == yEnd-1)
289             % if length of edge is one, the knot vector has got
```

```
290             % to be extended before a new edge can be added
291             obj.insertHorizontalEdge([xStart, xEnd], yEnd,false);
292
293             % update unrefined vertices from old indexing
294             update = find([zeros(i,1);...
295                 toRefine((i+1):end,2) >= yEnd]);
296             toRefine(update,2) = toRefine(update,2) + 1;
297         else
298             yCStart = obj.knotsYGlobal(yStart);
299             yCEnd = obj.knotsYGlobal(yEnd);
300             yCMiddle = 0.5*(yCStart+yCEnd);
301             yMiddle = find(obj.knotsYGlobal == yCMiddle);
302             assert(length(yMiddle) == 1);
303             obj.addSingleHorizontalEdge([xStart, xEnd],yMiddle);
304         end
305     end
306 end
307 end
```

A.1.4. Privat: Funktionen für Anker und lokale Indexvetoren

Listing A.14 (TMesh.m: (private) createAnchors).

```
309 function [result, type] = createAnchors(obj)
310     p1_even = (mod(obj.degree(1),2) == 0);
311     p1_odd = ~p1_even;
312     p2_even = (mod(obj.degree(2),2) == 0);
313     p2_odd = ~p2_even;
314     xSize = length(obj.knotsXGlobal);
315     ySize = length(obj.knotsYGlobal);
316     X = (1:xSize)'.*ones(1, ySize);
317     Y = ones(xSize,1).*(1:ySize);
318
319     % p1 & p2 odd: all active vertices (x)
320     if(p1_odd && p2_odd)
321         type = 'x';
322         result = obj.vertices(:,:);
323         % remove all non-active
324         result = and(result, obj.isActiveRegion(X,Y));
325     end
326
327     % p1 even, p2 odd: all active horizontal edges (h)
328     if(p1_even && p2_odd)
329         type = 'h';
330         sp = obj.hEdges(:,:) > 0;
331         isStartpointAR = obj.isActiveRegion(X.*sp, Y.*sp);
332         endpointsX = (X+obj.hEdges(:,:)).*sp;
333         endpointsY = Y.*sp;
334         isEndpointAR = obj.isActiveRegion(endpointsX, endpointsY);
335         result = and(isStartpointAR, isEndpointAR);
336     end
```

```
337
338     % p1 odd, p2 even: all active vertical edges (v)
339     if(p1_odd && p2_even)
340         type = 'v';
341         sp = obj.vEdges(:,:) > 0;
342         isStartpointAR = obj.isActiveRegion(X.*sp, Y.*sp);
343         endpointsX = X.*sp;
344         endpointsY = (Y + obj.vEdges(:,:)).*sp;
345         isEndpointAR = obj.isActiveRegion(endpointsX, endpointsY);
346         result = and(isStartpointAR, isEndpointAR);
347     end
348
349     % p1 & p2 even: all active tiles (t)
350     if(p1_even && p2_even)
351         type = 't';
352         sp = (obj.tiles(:,:,1) > 0);
353         isStartpointAR = obj.isActiveRegion(X.*sp,Y.*sp);
354         endpointsX = (X + obj.tiles(:,:,1)).*sp;
355         endpointsY = (Y + obj.tiles(:,:,2)).*sp;
356         isEndpointAR = obj.isActiveRegion(endpointsX, endpointsY);
357         result = and(isStartpointAR, isEndpointAR);
358     end
359 end
```

Listing A.15 (TMesh.m: (private) getHIndexVector).

```
362 function result = getHIndexVector(obj, a)
363     if(length(a)==1)
364         result = find(or(obj.vEdges(:,a) ~= 0,...
365             obj.vertices(:,a) == 1));
366     else
367         assert(a(2)>a(1));
368         result = find(sum(obj.vEdges(:,a(1):(a(2)-1)) ~= 0,2)...
369             == a(2)-a(1));
370     end
371 end
```

Listing A.16 (TMesh.m: (private) getLocalHIndexVector).

```
373 function result = getLocalHIndexVector(obj, a, b)
374     % result = getLocalHIndexVector(obj, a, b)
375     p1 = obj.degree(1);
376     isEven = (mod(p1,2) == 0);
377     vec = obj.getHIndexVector(b);
378     if(isEven)
379         assert(length(a) == 2);
380         floorMidIndex = find(vec == a(1));
381         ceilMidIndex = find(vec == a(2));
382         beginIndex = floorMidIndex - p1/2;
383         endIndex = ceilMidIndex + p1/2;
384         result = vec([beginIndex:floorMidIndex,...
```

```
385             ceilMidIndex:endIndex]);
386     else
387         assert(length(a) == 1);
388         middleIndex = find(vec == a);
389         beginIndex = middleIndex-(p1+1)/2;
390         endIndex =  middleIndex+(p1+1)/2;
391         result = vec(beginIndex:endIndex);
392     end
393 end
```

Listing A.17 (TMesh.m: (private) getVIndexVector).

```
395 function result = getVIndexVector(obj, a)
396     if(length(a)==1)
397         result = find(or(obj.hEdges(a,:) ~= 0,...
398             obj.vertices(a,:) == 1));
399     else
400         assert(a(2)>a(1));
401         result = find(sum(obj.hEdges(a(1):(a(2)-1),:) ~= 0,1)...
402             == a(2)-a(1));
403     end
404 end
```

Listing A.18 (TMesh.m: (private) getLocalVIndexVector).

```
406 function result = getLocalVIndexVector(obj, a, b)
407     p2 = obj.degree(2);
408     isEven = (mod(p2,2) == 0);
409     vec = obj.getVIndexVector(a);
410     if(isEven)
411         assert(length(b) == 2);
412         floorMidIndex = find(vec == b(1));
413         ceilMidIndex = find(vec == b(2));
414         beginIndex = floorMidIndex - p2/2;
415         endIndex = ceilMidIndex + p2/2;
416         result = vec([beginIndex:floorMidIndex,...
417             ceilMidIndex:endIndex]);
418     else
419         assert(length(b) == 1);
420         middleIndex = find(vec == b);
421         beginIndex = middleIndex-(p2+1)/2;
422         endIndex =  middleIndex+(p2+1)/2;
423         result = vec(beginIndex:endIndex);
424     end
425 end
```

Listing A.19 (TMesh.m: (private) isActiveRegion).

```
427 function result = isActiveRegion(obj, x, y)
428     n1_floor = 1;
429     n1_ceil = length(obj.knotsXGlobal);
430     p1 = obj.degree(1);
431     n2_floor = 1;
432     n2_ceil = length(obj.knotsYGlobal);
433     p2 = obj.degree(2);
434     % see section 2.1 of bachelor thesis
435     ar_x_floor = n1_floor + floor((p1+1)*0.5);
436     ar_x_ceil = n1_ceil - floor((p1+1)*0.5);
437     ar_y_floor = n2_floor + floor((p2+1)*0.5);
438     ar_y_ceil = n2_ceil - floor((p2+1)*0.5);
439     % result
440     erg_x = and(x >= ar_x_floor, x <= ar_x_ceil);
441     erg_y = and(y >= ar_y_floor, y <= ar_y_ceil);
442     result = and(erg_x, erg_y);
443 end
```

Listing A.20 (TMesh.m: (private) isFrameRegion).

```
445 function result = isFrameRegion(obj, x, y)
446     n1_floor = 1;
447     n1_ceil = length(obj.knotsXGlobal);
448     p1 = obj.degree(1);
449     n2_floor = 1;
450     n2_ceil = length(obj.knotsYGlobal);
451     p2 = obj.degree(2);
452     % see section 2.1 of bachelor thesis
453     ar_x_floor = n1_floor + floor((p1+1)*0.5);
454     ar_x_ceil = n1_ceil - floor((p1+1)*0.5);
455     ar_y_floor = n2_floor + floor((p2+1)*0.5);
456     ar_y_ceil = n2_ceil - floor((p2+1)*0.5);
457     % check if in x-margin or y-margin
458     x_margin = or(x<=ar_x_floor, x>=ar_x_ceil);
459     y_margin = or(y<=ar_y_floor, y>=ar_y_ceil);
460     result = or(x_margin, y_margin);
461 end
```

A.1.5. Privat: Funktionen zur T-Gitter-Manipulation

Knotenvektoren

Listing A.21 (TMesh.m: (private) insertKnotX).

```
464 function obj = insertKnotX(obj, x)
465     % index-based insertation of knot in x-direction, adaptation
466     % of the mesh accordingly
467     assert(x ~= 1);
468
469     % updating the knot-vector
```

```
470     xCoord = 0.5*(obj.knotsXGlobal(x-1)+obj.knotsXGlobal(x));
471     ySize = length(obj.knotsYGlobal);
472     obj.knotsXGlobal = [ obj.knotsXGlobal(1:(x-1)) xCoord,...
473         obj.knotsXGlobal(x:end)];
474
475     % updating the vertices
476     obj.vertices = [obj.vertices(1:(x-1),:); zeros(1,ySize);...
477         obj.vertices(x:end,:)];
478
479     % updating edges and tiles
480     obj.vEdges = [obj.vEdges(1:(x-1),:); zeros(1,ySize); ...
481         obj.vEdges(x:end,:)];
482     obj.hEdges = [obj.hEdges(1:(x-1),:); zeros(1,ySize); ...
483         obj.hEdges(x:end,:)];
484     obj.tiles = [obj.tiles(1:(x-1),:,:); zeros(1,ySize,3); ...
485         obj.tiles(x:end,:,:)];
486
487     % updating all entries that point to a changed edge/ tile
488     for i = 1:ySize
489         if(obj.isOnHEdge(x-1, i))
490           [edgeX, edgeY] = obj.findHEdgeIndex(x-1,i);
491           obj.hEdges(edgeX,edgeY) = obj.hEdges(edgeX,edgeY)+1;
492           obj.refreshHEdgeInfo(edgeX, edgeY);
493           if(i~=ySize)
494             [tileX, tileY] = obj.findTileIndex(x-1,i);
495             obj.tiles(tileX,tileY,1) = obj.tiles(tileX,tileY,1)+1;
496             obj.refreshTileInfo(tileX, tileY);
497           end
498         end
499     end
500 end
```

Listing A.22 (TMesh.m: (private) insertKnotY).

```
502 function obj = insertKnotY(obj, y)
503     % index-based insertation of knot in y-direction, adaptation
504     % of the mesh accordingly
505     assert(y ~= 1);
506
507     % updating the knot-vector
508     yCoord = 0.5*(obj.knotsYGlobal(y-1)+obj.knotsYGlobal(y));
509     xSize = length(obj.knotsXGlobal);
510     obj.knotsYGlobal = [ obj.knotsYGlobal(1:(y-1)) yCoord,...
511         obj.knotsYGlobal(y:end)];
512
513     % updating the vertices
514     obj.vertices = [obj.vertices(:,1:(y-1)), zeros(xSize,1),...
515         obj.vertices(:,y:end)];
516
517     % updating edges and tiles
518     obj.vEdges = [obj.vEdges(:,1:(y-1)), zeros(xSize,1),...
519         obj.vEdges(:,y:end)];
```

```
520     obj.hEdges = [obj.hEdges(:,1:(y-1)), zeros(xSize,1),...
521         obj.hEdges(:,y:end)];
522     obj.tiles = [obj.tiles(:,1:(y-1),:), zeros(xSize,1,3),...
523         obj.tiles(:,y:end,:)];
524
525     % updating all entries that point to a changed edge/ tile
526     for i = 1:xSize
527         if(obj.isOnVEdge(i,y-1))
528           [edgeX, edgeY] = obj.findVEdgeIndex(i,y-1);
529           obj.vEdges(edgeX,edgeY) = obj.vEdges(edgeX,edgeY)+1;
530           obj.refreshVEdgeInfo(edgeX, edgeY);
531           if(i~=xSize)
532             [tileX, tileY] = obj.findTileIndex(i,y-1);
533             obj.tiles(tileX,tileY,2) = obj.tiles(tileX,tileY,2)+1;
534             obj.refreshTileInfo(tileX, tileY);
535           end
536         end
537     end
538 end
```

Listing A.23 (TMesh.m: (private) assureOpenKnotVectors).

```
540 function obj = assureOpenKnotVectors(obj)
541     p = obj.degree(1);
542     q = obj.degree(2);
543     x_zeros = sum(obj.knotsXGlobal == 0);
544     for i = x_zeros:p
545         obj.insertKnotX(2);
546         obj.knotsXGlobal(2) = 0;
547         obj.addSingleVerticalEdge(2,[1,length(obj.knotsYGlobal)]);
548     end
549     x_ones = sum(obj.knotsXGlobal == 1);
550     for i = x_ones:p
551         len = length(obj.knotsXGlobal);
552         obj.insertKnotX(len);
553         obj.knotsXGlobal(len) = 1;
554         obj.addSingleVerticalEdge(len,[1,length(obj.knotsYGlobal)]);
555     end
556     y_zeros = sum(obj.knotsYGlobal == 0);
557     for i = y_zeros:q
558         obj.insertKnotY(2);
559         obj.knotsYGlobal(2) = 0;
560         obj.addSingleHorizontalEdge([1,length(obj.knotsXGlobal)],2);
561     end
562     y_ones = sum(obj.knotsYGlobal == 1);
563     for i = y_ones:q
564         len = length(obj.knotsYGlobal);
565         obj.insertKnotY(len);
566         obj.knotsYGlobal(len) = 1;
567         obj.addSingleHorizontalEdge([1,length(obj.knotsXGlobal)],len);
568     end
569 end
```

Eckpunkte

Listing A.24 (TMesh.m: (private) addVertex).

```
572 function obj = addVertex(obj, x, y)
573     % addVertex(obj, xCoord, yCoord)
574     %   (index-based): adds the given vertex to the mesh
575     assert(length(x)==length(y));
576     assert(sum(floor(x))==sum(x));
577     assert(sum(floor(y))==sum(y));
578     for i = 1:length(x)
579         assert(obj.isOnEdge(x(i),y(i)));
580         if(obj.isVertex(x(i), y(i))==false)
581             obj.vertices(x(i), y(i))=1;
582             obj.splitEdge(x(i), y(i));
583         end
584     end
585 end
```

Listing A.25 (TMesh.m: (private) isVertex).

```
587 function result = isVertex(obj, x, y)
588     % res = isVertex(x, y)
589     %   (index-based): Returns true, if the given coordinates
590     %   represent a vertex and false otherwise.
591     result = (obj.vertices(x,y)==1);
592 end
```

Kanten

Listing A.26 (TMesh.m: (private) splitEdge).

```
595 function obj = splitEdge(obj, x, y)
596     assert(obj.isOnEdge(x,y));
597     [startX, startY] = obj.findHEdgeIndex(x,y);
598     if(startX > 0)
599         oldLength = obj.hEdges(startX, startY);
600         adjustedLength = x - startX;
601         obj.hEdges(startX, startY) = adjustedLength;
602         obj.hEdges(x,y) = oldLength-adjustedLength;
603         obj.refreshHEdgeInfo(startX, startY);
604         obj.refreshHEdgeInfo(x,y);
605     else
606         [startX, startY] = obj.findVEdgeIndex(x,y);
607         assert(startX > 0, "No edge found to be splitted");
608         oldLength = obj.vEdges(startX, startY);
609         adjustedLength = y - startY;
610         obj.vEdges(startX, startY) = adjustedLength;
611         obj.vEdges(x,y) = oldLength-adjustedLength;
612         obj.refreshVEdgeInfo(startX, startY);
613         obj.refreshVEdgeInfo(x,y);
614     end
615 end
```

Listing A.27 (TMesh.m: (private) isOnEdge).

```
617 function result = isOnEdge(obj, x, y)
618     % isOnEdge(x, y)
619     %   (index-based): Returns true, if the given coordinates
620     %   are element of an edge and false otherwise.
621     isOnHEdge = (obj.hEdges(x,y) ~= 0);
622     isOnVEdge = (obj.vEdges(x,y) ~= 0);
623     result = (isOnHEdge || isOnVEdge);
624 end
```

Vertikale Kanten

Listing A.28 (TMesh.m: (private) insertVerticalEdge).

```
627 function obj = insertVerticalEdge(obj, xIndex, yEdge)
628     % (index-based) Extends the knot-vector in x-direction at
629     % position xIndex, adds two vertices at xIndex/yEdge(1) and
630     % xIndex/yEdge(2) and connects them.
631
632     if(nargin < 3)
633         error("Not enough input arguments");
634     end
635     obj.insertKnotX(xIndex);
636     obj.addSingleVerticalEdge(xIndex, yEdge);
637 end
```

Listing A.29 (TMesh.m: (private) addSingleVerticalEdge).

```
639 function obj = addSingleVerticalEdge(obj, x, yEdge)
640     % addSingleVerticalEdge(obj, x, yEdge)
641     %   (index-based)
642     assert(length(yEdge)==2, "Exactly two indices necessary.");
643     yA = yEdge(1);
644     yB = yEdge(2);
645     assert(obj.isVertex(x,yA) && obj.isVertex(x,yB));
646     for i = (yA+1):(yB-1)
647         assert(obj.vertices(x,i)==0, "New vertical edge" +...
648             " is intersecting vertex at [" + x + "," + i + "]!");
649         assert(obj.vEdges(x,i)==0, "New vertical edge is " +...
650             "intersecting hor. edge at [" + x + "," + i + "]!");
651     end
652     [xTileStart, yTileStart] = obj.findTileIndex(x-1, yA);
653     xTileEnd = xTileStart + obj.tiles(xTileStart, yTileStart, 1);
654     yTileEnd = yTileStart + obj.tiles(xTileStart, yTileStart, 2);
655
656     % double-check: yA == yTileStart, yB == yTileEnd !
657     assert(yA == yTileStart && yB == yTileEnd);
658     if(obj.tiles(xTileStart, yTileStart, 3) == -1)
659         newTileLevel = -1;
660     else
661         assert(mod(obj.tiles(xTileStart, yTileStart, 3),1) == 0);
```

```
662             newTileLevel = obj.tiles(xTileStart, yTileStart, 3) + 0.5;
663         end
664
665         % reduce the existing tile, only value of x has to be adapted
666         obj.tiles(xTileStart, yTileStart, 1) = x - xTileStart;
667         obj.tiles(xTileStart, yTileStart, 3) = newTileLevel;
668
669         % add a new tile and update tile info within tile
670         obj.tiles(x,yA,1) = xTileEnd-x;
671         obj.tiles(x,yA,2) = yTileEnd-yA;
672         obj.tiles(x,yA,3) = newTileLevel;
673         obj.refreshTileInfo(x,yA);
674
675         % add the edge and update edge info
676         obj.vEdges(x,yA) = yB-yA;
677         obj.refreshVEdgeInfo(x,yA);
678     end
```

Listing A.30 (TMesh.m: (private) refreshVEdgeInfo).

```
680 function obj = refreshVEdgeInfo(obj, x, y)
681     ySize = obj.vEdges(x,y);
682     % see refreshTileInfo();
683     yMarginExpansion = 1*((y+ySize)==length(obj.knotsYGlobal));
684     for i = 1:(ySize-1+yMarginExpansion)
685         obj.vEdges(x, y+i) = -i;
686     end
687 end
```

Listing A.31 (TMesh.m: (private) findVEdgeIndex).

```
689 function [xResultIndex, yResultIndex] = findVEdgeIndex(obj, x, y)
690     edgeVal = obj.vEdges(x,y);
691     if(edgeVal > 0)
692         xResultIndex = x;
693         yResultIndex = y;
694     else
695         if(edgeVal < 0)
696             xResultIndex = x;
697             yResultIndex = y + edgeVal;
698         else
699             xResultIndex = -1;
700             yResultIndex = -1;
701         end
702     end
703 end
```

Listing A.32 (TMesh.m: (private) isOnVEdge).

```
705 function result = isOnVEdge(obj, x, y)
706     % isOnVEdge(x, y)
707     %   (index-based): Returns true, if the given coordinates
708     %   are element of an vertical edge and false otherwise.
709     result = (obj.vEdges(x,y) ~= 0);
710 end
```

Horizontale Kanten

Listing A.33 (TMesh.m: (private) insertHorizontalEdge).

```
713 function obj = insertHorizontalEdge(obj, xEdge, yIndex)
714     % (index-based) Extends the knot-vector in y-direction at
715     % position yIndex, adds two vertices at xEdge(1)/yIndex and
716     % xEdge(2)/yIndex and connects them.
717 
718     if(nargin < 3)
719         error("Not enough input arguments");
720     end
721     obj.insertKnotY(yIndex);
722     obj.addSingleHorizontalEdge(xEdge, yIndex);
723 end
```

Listing A.34 (TMesh.m: (private) addSingleHorizontalEdge).

```
725 function obj = addSingleHorizontalEdge(obj, xEdge, y)
726     % addSingleHorizontalEdge(obj, xEdge, y)
727     %   (index-based)
728     assert(length(yEdge)==2, "Exactly two indices necessary.");
729     xA = xEdge(1);
730     xB = xEdge(2);
731     assert(obj.isVertex(xA,y) && obj.isVertex(xB,y));
732     for i = (xA+1):(xB-1)
733         assert(obj.vertices(i,y)==0, "New horizontal edge"+...
734             " is intersecting vertex at [" + i + "," + y + "!");
735         assert(obj.vEdges(i,y)==0, "New horizontal edge is" + ...
736             " intersecting vert. edge at [" + i + "," + y + "!");
737     end
738     [xTileStart, yTileStart] = obj.findTileIndex(xA, y-1);
739     xTileEnd = xTileStart + obj.tiles(xTileStart, yTileStart, 1);
740     yTileEnd = yTileStart + obj.tiles(xTileStart, yTileStart, 2);
741 
742     % double-check: xA == xTileStart, xB == xTileEnd !
743     assert(xA == xTileStart && xB == xTileEnd);
744     if(obj.tiles(xTileStart, yTileStart, 3) == -1)
745         newTileLevel = -1;
746     else
747         obj.tiles(xTileStart, yTileStart, 3);
748         mod(obj.tiles(xTileStart, yTileStart, 3),1);
749         assert(mod(obj.tiles(xTileStart, yTileStart, 3),1) == 0.5);
750         newTileLevel = obj.tiles(xTileStart, yTileStart, 3) + 0.5;
751     end
```

```
752
753     % reduce the existing tile, only value of y has to be adapted
754     obj.tiles(xTileStart, yTileStart, 2) = y - yTileStart;
755     obj.tiles(xTileStart, yTileStart, 3) = newTileLevel;
756
757     % add a new tile and update tile info within tile
758     obj.tiles(xA,y,1) = xTileEnd-xA;
759     obj.tiles(xA,y,2) = yTileEnd-y;
760     obj.tiles(xA,y,3) = newTileLevel;
761     obj.refreshTileInfo(xA,y);
762
763     % add the edge and update edge info
764     obj.hEdges(xA,y) = xB-xA;
765     obj.refreshHEdgeInfo(xA,y);
766 end
```

Listing A.35 (TMesh.m: (private) refreshHEdgeInfo).

```
768 function obj = refreshHEdgeInfo(obj, x, y)
769     xSize = obj.hEdges(x,y);
770     % see refreshTileInfo();
771     xMarginExpansion = 1*((x+xSize)==length(obj.knotsXGlobal));
772     for i = 1:(xSize-1+xMarginExpansion)
773         obj.hEdges(x+i, y) = -i;
774     end
775 end
```

Listing A.36 (TMesh.m: (private) findHEdgeIndex).

```
777 function [xResultIndex, yResultIndex] = findHEdgeIndex(obj, x, y)
778     edgeVal = obj.hEdges(x,y);
779     if(edgeVal > 0)
780         xResultIndex = x;
781         yResultIndex = y;
782     else
783         if(edgeVal < 0)
784             xResultIndex = x + edgeVal;
785             yResultIndex = y;
786         else
787             xResultIndex = -1;
788             yResultIndex = -1;
789         end
790     end
791 end
```

Listing A.37 (TMesh.m: (private) isOnHEdge).

```
793 function result = isOnHEdge(obj, x, y)
794     % isOnHEdge(x, y)
```

```
795     %   (index-based): Returns true, if the given coordinates
796     %   are element of an horizontal edge and false otherwise.
797     result = (obj.hEdges(x,y) ~= 0);
798 end
```

Elemente des T-Gitters

Listing A.38 (TMesh.m: (private) findTileIndex).

```
801 function [xResultIndex, yResultIndex] = findTileIndex(obj, x, y)
802     % [xResultIndex, yResultIndex] = findTileIndex(obj, x, y)
803     %   (index-based)
804     if(obj.tiles(x,y, 1) > 0)
805         xResultIndex = x;
806         yResultIndex = y;
807     else
808         xResultIndex = x + obj.tiles(x,y, 1);
809         yResultIndex = y + obj.tiles(x,y, 2);
810     end
811 end
```

Listing A.39 (TMesh.m: (private) refreshTileInfo).

```
813 function obj = refreshTileInfo(obj, x, y)
814     xSize = obj.tiles(x,y,1);
815     ySize = obj.tiles(x,y,2);
816     % writes on upper resp. right edge if tile is on upper resp.
817     % right margin. "Next knot" is on the end of global knots.
818     xMarginExpansion = 1*((x+xSize)==length(obj.knotsXGlobal));
819     yMarginExpansion = 1*((y+ySize)==length(obj.knotsYGlobal));
820     for i = 0:(xSize-1+xMarginExpansion)
821         for j = 0:(ySize-1+yMarginExpansion)
822             % write x- and y-distance in every field except start
823             if(i~=0 || j~=0)
824                 obj.tiles(x+i, y+j, 1) = -i;
825                 obj.tiles(x+i, y+j, 2) = -j;
826             end
827         end
828     end
829 end
```

A.2. IGAFEM.m

Listing B.40 (IGAFEM.m: IGAFEM).

```
1 function [mesh, error, energy] = IGAFEM(mesh, f, doerflerTheta)
2
3 [anchors, xKnots, yKnots] = mesh.getAnchors();
```

```
4  [xGlobalKnots, yGlobalKnots] = mesh.getUniqueGlobalKnots();
5  [p,q] = mesh.getPolynomialDegree();
6  int_degree = max([p,q])*2;
7
8  [weights, nodes] = gaussLegendre(0,1,int_degree);
9
10 n = size(anchors,1);
11
12 %------- SOLVE ----------------------
13
14 % fulfilling homogenous dirichlet-boundary conditions
15 to_skip = (zeros(n,1) == 1);
16 for i = 1:n
17     x_mult_zero = sum(xKnots(i,:) == 0);
18     x_mult_one = sum(xKnots(i,:) == 1);
19     y_mult_zero = sum(yKnots(i,:) == 0);
20     y_mult_one = sum(yKnots(i,:) == 1);
21     to_skip(i) = (x_mult_zero == p+1 || x_mult_one == p+1 || ...
22         y_mult_zero == q+1 || y_mult_one == q+1);
23 end
24 inner_indices = find(~to_skip)';
25
26 % Building Galerkins matrix A
27 A = zeros(n);
28 for i = inner_indices
29     for j = inner_indices(inner_indices >= i)
30         x_floor = max(xKnots([i,j],1));
31         x_ceil = min(xKnots([i,j],end));
32         y_floor = max(yKnots([i,j],1));
33         y_ceil = min(yKnots([i,j],end));
34         if(x_floor < x_ceil && y_floor < y_ceil)
35
36             % x: gauss-legendre
37             x_floor_index = find(xGlobalKnots == x_floor);
38             x_ceil_index = find(xGlobalKnots == x_ceil);
39             y_floor_index = find(yGlobalKnots == y_floor);
40             y_ceil_index = find(yGlobalKnots == y_ceil);
41
42             res = 0;
43             for k = x_floor_index:(x_ceil_index-1)
44                 a_x = xGlobalKnots(k);
45                 b_x = xGlobalKnots(k+1);
46                 for m = y_floor_index:(y_ceil_index-1)
47                     a_y = yGlobalKnots(m);
48                     b_y = yGlobalKnots(m+1);
49                     [xNodes, yNodes] = meshgrid(nodes*...
50                         (b_x-a_x)+a_x, nodes*(b_y-a_y)+a_y);
51
52                     % partially deriven splines
53                     B_i_x = evalBDerive(xKnots(i,:),p,xNodes).*...
54                         evalBspline(yKnots(i,:),q,yNodes);
55                     B_i_y = evalBspline(xKnots(i,:),p,xNodes).*...
56                         evalBDerive(yKnots(i,:),q,yNodes);
```

```
57                      B_j_x = evalBDerive(xKnots(j,:),p,xNodes).*...
58                          evalBspline(yKnots(j,:),q,yNodes);
59                      B_j_y = evalBspline(xKnots(j,:),p,xNodes).*...
60                          evalBDerive(yKnots(j,:),q,yNodes);
61
62                      % cumulated
63                      nablaB = B_i_x.*B_j_x + B_i_y.*B_j_y;
64                      mWeight = (b_y-a_y)*(b_x-a_x)*weights'.*weights;
65                      res = res + sum(mWeight.*(nablaB),'all');
66                  end
67              end
68              A(j,i) = res;
69          end
70      end
71      for j = inner_indices(inner_indices < i)
72          if(A(i,j) ~= 0)
73              A(j,i) = A(i,j);
74          end
75      end
76  end
77
78  % Building F
79  F = zeros(n,1);
80  for i = inner_indices
81      x_floor = xKnots(i,1);
82      x_ceil = xKnots(i,end);
83      y_floor = yKnots(i,1);
84      y_ceil = yKnots(i,end);
85
86      x_floor_index = find(xGlobalKnots == x_floor);
87      x_ceil_index = find(xGlobalKnots == x_ceil);
88      y_floor_index = find(yGlobalKnots == y_floor);
89      y_ceil_index = find(yGlobalKnots == y_ceil);
90      sol = 0;
91      for j = x_floor_index:(x_ceil_index-1)
92          a_x = xGlobalKnots(j);
93          b_x = xGlobalKnots(j+1);
94          for k = y_floor_index:(y_ceil_index-1)
95              a_y = yGlobalKnots(k);
96              b_y = yGlobalKnots(k+1);
97              [xNodes, yNodes] = meshgrid(nodes*(b_x-a_x)+a_x,...
98                  nodes*(b_y-a_y)+a_y);
99
100             mWeights = (b_y-a_y)*(b_x-a_x)*weights'.*weights;
101             xEval = evalBspline(xKnots(i,:),p,xNodes);
102             yEval = evalBspline(yKnots(i,:),q,yNodes);
103             sol = sol + sum(mWeights.*f(xNodes, yNodes).*...
104                 xEval.*yEval,'all');
105         end
106     end
107     F(i) = sol;
108 end
109
```

```
110 F(to_skip) = [];
111 A(to_skip,:) = [];
112 A(:,to_skip) = [];
113
114 x_short = A\F;
115 x = zeros(n,1);
116 x(~to_skip) = x_short;
117 energy = x_short'*A*x_short;
118 controlPoints = [anchors, x];
119
120 %------- ESTIMATE -------------------
121 tiles = mesh.getNonzeroActiveTiles();
122 eta = zeros(size(tiles,1),1);
123
124 for i = 1:size(tiles,1)
125     x1 = tiles(i,1);
126     x2 = tiles(i,3);
127     y1 = tiles(i,2);
128     y2 = tiles(i,4);
129
130     x_floor_index = find(xGlobalKnots == x1);
131     x_ceil_index = find(xGlobalKnots == x2);
132     y_floor_index = find(yGlobalKnots == y1);
133     y_ceil_index = find(yGlobalKnots == y2);
134     sol = 0;
135     for j = x_floor_index:(x_ceil_index-1)
136         a_x = xGlobalKnots(j);
137         b_x = xGlobalKnots(j+1);
138         for k = y_floor_index:(y_ceil_index-1)
139             a_y = yGlobalKnots(k);
140             b_y = yGlobalKnots(k+1);
141             [xNodes, yNodes] = meshgrid(nodes*(b_x-a_x)+a_x,...
142                 nodes*(b_y-a_y)+a_y);
143
144             % partially deriven splines
145             [B_xx, B_yy] = evalULaplace(xNodes,...
146                 yNodes, controlPoints(:,3), xKnots, yKnots, p, q);
147
148             evaledTile = (B_xx + B_yy + f(xNodes, yNodes)).*...
149                 (B_xx + B_yy + f(xNodes, yNodes));
150             mWeights = (b_y-a_y)*(b_x-a_x)*weights'.*weights;
151             sol = sol + sum(mWeights.*evaledTile,'all');
152         end
153     end
154
155     areaQ = (y2-y1)*(x2-x1);
156     eta(i) = areaQ*sol;
157 end
158
159 tot_error = sqrt(sum(eta));
160 error(1,[1,2]) = [size(tiles,1), tot_error];
161
162 %------- MARK -----------------------
```

```
163 [~, sortingIndex] = sort(eta,'descend');
164 sorted_eta = eta(sortingIndex);
165 marked_error = 0;
166 num_el = 0;
167 if(doerflerTheta < 1)
168     while(marked_error < doerflerTheta*tot_error*tot_error)
169         num_el = num_el + 1;
170         marked_error = marked_error + sorted_eta(num_el);
171     end
172     refinementIndices = sortingIndex(1:num_el);
173     refinementTiles = tiles(sort(refinementIndices),[1,2]);
174 else
175     refinementTiles = tiles(:,[1,2]);
176 end
177
178 %------- REFINE ----------------------
179 mesh = mesh.refineTile(refinementTiles);
180 end
```

Listing B.41 (IGAFEM.M: EVALULAPLACE).

```
182 function [laplaceX, laplaceY] = evalULaplace...
183     (X, Y, anchorPoints, xKnots, yKnots, p, q)
184 assert(size(X,1) == size(Y,1) && size(X,2) == size(Y,2))
185 laplaceX = 0*X;
186 laplaceY = 0*X;
187 nonZeroAnchors = find(anchorPoints ~= 0);
188 for i = nonZeroAnchors'
189     xTBE = and(xKnots(i,1)<=X, xKnots(i,end)>=X);
190     yTBE = and(yKnots(i,1)<=Y, yKnots(i,end)>=Y);
191     toEval = and(xTBE,yTBE);
192
193     xSpline = evalBspline(xKnots(i,:),p,X(toEval));
194     ySpline = evalBspline(yKnots(i,:),q,Y(toEval));
195     xDeriv2 = evalBLaplace(xKnots(i,:),p,X(toEval));
196     yDeriv2 = evalBLaplace(yKnots(i,:),q,Y(toEval));
197
198     laplaceX(toEval) = laplaceX(toEval) + ...
199         xDeriv2.*ySpline*anchorPoints(i);
200     laplaceY(toEval) = laplaceY(toEval) + ...
201         xSpline.*yDeriv2*anchorPoints(i);
202 end
203 end
```

Listing B.42 (IGAFEM.M: EVALBDERIVE).

```
205 function res = evalBDerive(knots,p,x)
206 assert(length(knots) >= p+2);
207 if(nargin < 3)
208     error("Not enough input arguments");
209 end
```

```
210 res = x*0;
211 if(p ~= 0)
212     if(knots(1+p)-knots(1) ~= 0)
213         b_i_pMinus = evalBspline(knots,p-1,x,1);
214         res = res + (p/(knots(1+p)-knots(1)))*b_i_pMinus;
215     end
216     if(knots(2+p)-knots(2) ~= 0)
217         b_iPlus_pMinus = evalBspline(knots,p-1,x,2);
218         res = res - (p/(knots(2+p)-knots(2)))*b_iPlus_pMinus;
219     end
220 end
221 end
```

Listing B.43 (IGAFEM.m: evalBLaplace).

```
223 function res = evalBLaplace(knots,p,x)
224 assert(length(knots) >= p+2);
225 if(nargin < 3)
226     error("Not enough input arguments");
227 end
228 res = x*0;
229 if(p > 1)
230     if(knots(1+p)-knots(1) ~= 0)
231         if(knots(p)-knots(1) ~= 0)
232             b_i_pMinusTwo = evalBspline(knots,p-2,x,1);
233             res = res + (p/(knots(1+p)-knots(1)))*((p-1)/...
234                 (knots(p)-knots(1)))*b_i_pMinusTwo;
235         end
236         if(knots(1+p)-knots(2) ~= 0)
237             b_iPlus_pMinusTwo = evalBspline(knots,p-2,x,2);
238             res = res - (p/(knots(1+p)-knots(1)))*((p-1)/...
239                 (knots(1+p)-knots(2)))*b_iPlus_pMinusTwo;
240         end
241     end
242     if(knots(2+p)-knots(2) ~= 0)
243         if(knots(1+p)-knots(2) ~= 0)
244             b_iPlus_pMinusTwo = evalBspline(knots,p-2,x,2);
245             res = res - (p/(knots(2+p)-knots(2)))*((p-1)/...
246                 (knots(1+p)-knots(2)))*b_iPlus_pMinusTwo;
247         end
248         if(knots(2+p)-knots(3) ~= 0)
249             b_iPlusTwo_pMinusTwo = evalBspline(knots,p-2,x,3);
250             res = res + (p/(knots(2+p)-knots(2)))*((p-1)/...
251                 (knots(2+p)-knots(3)))*b_iPlusTwo_pMinusTwo;
252         end
253     end
254 end
255 end
```

Listing B.44 (IGAFEM.m: gaussLegendre).

```
257 % Gauss-Legendre-Quadrature
258 function [weights, nodes] = gaussLegendre(a, b, degree)
259 n = ceil((degree-1)/2);
260
261 orth_pol = eye(n+2);
262 for i = 1:(n+2)
263     for j = 1:(i-1)
264         inner = innerProduct(orth_pol(i,:),orth_pol(j,:),a,b);
265         orth_pol(i,:) = orth_pol(i,:)-inner*orth_pol(j,:);
266     end
267     abs_val = innerProduct(orth_pol(i,:),orth_pol(i,:),a,b);
268     orth_pol(i,:) = orth_pol(i,:)/sqrt(abs_val);
269 end
270
271 nodes = roots(orth_pol(n+2,end:(-1):1));
272 nodes = sort(nodes);
273 nodes = reshape(nodes, 1, []);
274
275 lagrange = zeros(n+1);
276 weights = zeros(1,n+1);
277 for i = 1:(n+1)
278     lagrange(i,1) = 1;
279     for j = 1:(n+1)
280         if(i~=j)
281             lagrange(i,:) = polynomProduct(lagrange(i,1:(end-1)),...
282                 [-nodes(j), 1]/(nodes(i)-nodes(j)));
283         end
284     end
285     weights(i) = polynomInt(lagrange(i,:),a,b);
286 end
287 end
288
289 function res = polynomProduct(s,t)
290 res = zeros(1, length(s)+length(t)-1);
291 for i=1:length(t)
292     res(1,i:(length(s)+i-1)) = s.*t(i) + res(1,i:(length(s)+i-1));
293 end
294 end
295
296 function res = innerProduct(s, t, a, b)
297 resultingPolynom = polynomProduct(s,t);
298 res = polynomInt(resultingPolynom, a, b);
299 end
300
301 function res = polynomInt(coeff, a, b)
302 num_el = length(coeff);
303 coeff = reshape(coeff, 1, []).*(1./(1:num_el));
304 res = sum(coeff.*(b.^(1:num_el)-a.^(1:num_el)));
305 end
```

A.3. evalBSpline.c

Listing C.45 (evalBspline).

```
#include "mex.h"

void mexFunction(int nlhs, mxArray* plhs[],
        int nrhs, const mxArray* prhs[])
{
    if(nrhs < 3 || nrhs > 4)
        mexErrMsgTxt("Falsche Anzahl an Parametern!");
    double* xi = mxGetPr(prhs[0]); // vector of knots, xi
    int xi_n = mxGetM(prhs[0])*mxGetN(prhs[0]);
    int p = (int) mxGetPr(prhs[1])[0]; // polynomial degree
    double* xMatrix = mxGetPr(prhs[2]); // evaluation-points
    int M = mxGetM(prhs[2]);
    int N = mxGetN(prhs[2]);
    int offset = 0; // starting index/ offset
    if (nrhs == 4)
        offset = mxGetPr(prhs[3])[0]-1; // Matlab -> C-indexing
    if(p+offset+1 >= xi_n)
        mexErrMsgTxt("Knotenvektor Xi zu klein!");
    if(p < 0)
        mexErrMsgTxt("Polynomgrad p kleiner null!");

    plhs[0] = mxCreateDoubleMatrix(M,N,mxREAL);
    double* out = mxGetPr(plhs[0]);

    double* val = malloc((p+1)*sizeof(double));

    for(int k = 0; k < M*N; ++k) {
        double x = xMatrix[k];
        // first level
        for(int i = 0; i <= p; ++i) {
            if(x >= xi[i+offset] && x < xi[i+offset+1])
                val[i] = 1;
            else
                val[i] = 0;
        }

        for(int i = 1; i <= p; ++i) {
            for(int j = 0; j <= p-i; ++j) {
                double interim = 0;
                if(xi[j+offset] != xi[j+offset+i]) {
                    interim = interim + ((x-xi[j+offset])/
                            (xi[j+offset+i]-xi[j+offset]))*val[j];
                }
                if(xi[j+offset+1] != xi[j+offset+i+1]) {
                    interim = interim + ((xi[j+offset+i+1]-x)/
                            (xi[j+offset+i+1]-xi[j+offset+1]))*
                            val[j+1];
                }
                val[j] = interim;
```

```
50                  }
51              }
52              out[k] = val[0];
53          }
54          free(val);
55          val = NULL;
56      }
```