TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Bachelorarbeit

Vektorisierte Implementierung adaptiver P1-FEM in 3D

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

unter der Anleitung von
Ao.Univ.Prof. Dipl.Math. Dr.techn. Dirk Praetorius
Marcus Page, M.Sc.

durch
Josef Friedrich Kemetmüller
Marktplatz 13
4501 Neuhofen an der Krems

Inhaltsverzeichnis

0 Einleitung

Eines der wohl wichtigsten Gebiete der Mathematik, das direkt in anderen Disziplinen Anwendung findet ist zweifelsohne die Theorie der Differentialgleichungen. In fast allen naturwissenschaftlichen Feldern treten diese an gewissen Stellen auf. Da Differentialgleichungen allerdings oft nur schwer und in noch mehr Fällen gar nicht analytisch lösbar sind, bedient man sich bei deren Behandlung numerischer Lösungsverfahren.

0.1 Überblick

In der hier vorgestellten Arbeit wird auf die *Methode der finiten Elemente* („FEM“) eingegangen, die bei elliptischen Differentialgleichungen Anwendung findet. Das Verfahren zählt zur Klasse der Galerkin-Verfahren. Bei diesen wird das Problem in die *schwache Formulierung* auf einem Hilbertraum überführt und auf einem geeigneten endlichdimensionalen Teilraum mit Mitteln der linearen Algebra numerisch gelöst.

Dieses Konzept kann nun auf verschiedene Weisen realisiert werden. Im Zuge der Klärung der konkreten Herangehensweise dieser Arbeit schlüsseln wir hier den Titel auf: Am Modellbeispiel der Laplace-Gleichung werden MATLAB-Methoden der FEM vorgestellt. Als Definitionsbereich der Differentialgleichung wird von einem Polyeder im $\mathbb{R}^3$ („3D“) ausgegangen. Dieser wird in die finiten Elemente zerlegt, die hier als Tetraeder gewählt sind. Als endlichdimensionalen Raum wählen wir alle elementweise (auf jedem Tetraeder) affinen und global stetigen Funktionen. Die Wahl des Raums der Polynome vom Grad eins liefert also das „P1“ im Titel. „Vektorisierung“ ist ein in MATLAB übliches Konzept um allzu große Geschwindigkeitseinbußen zu vermeiden, die dadurch entstehen, dass MATLAB eine interpretierte Programmiersprache ist. Hierbei wird soweit möglich versucht Schleifen durch äquivalente Vektoroperationen zu ersetzen.

Zu guter Letzt klären wir die wohl wichtigste hier verwendete Herangehensweise: Die Adaptivität. Wie zuvor erwähnt sind die generierten Lösungen auf den Tetraedern der gewählten Zerlegung elementweise affin. Mit der Anzahl der Tetraeder erhöht sich also die Anzahl der Freiheitsgrade und damit sowohl der Rechenaufwand, als auch (vorerst zumindest anschaulich) die Güte der Lösung. Doch das wirft natürlich die Frage auf, welche Zerlegung des Gebietes man am besten verwenden sollte. Neben der Möglichkeit lediglich eine einzige Berechnung mit einer geeignet feinen Zerlegung durchzuführen, kann hier beispielsweise zuerst eine Berechnung mit einer groben Zerlegung stattfinden und iterativ durch Teilung jedes Tetraeders und erneuter Lösung, das Verfahren wiederholt werden bis die gewünschte Genauigkeit (oder die Grenze der vorhandenen Hardware) erreicht ist. Dieses Vorgehen nennt man auch *uniforme Verfeinerung*. Doch man stelle sich nun beispielsweise vor die gesuchte Lösung oszilliere stark auf einem kleinen Teil des Gebietes und verhalte sich annähernd konstant auf dem Komplement. Hier wäre es wünschenswert die verfügbare Rechenkapazität auf den stark oszillierenden Bereich zu konzentrieren, um diesem Ausschnitt mehr Aufmerksamkeit zu schenken. Mit dem zuvor erwähnten uniformen Verfahren hat man sein Pulver dabei schon auf dem restlichen Gebiet verschossen. Hier kommt das „adaptive“ Verfahren ins Spiel. Mit sogenannten *Fehlerschätzern* lässt sich – ohne die exakte Lösung zu kennen – ein ungefähres Maß für den Fehler auf den Tetraedern angeben. Die Tetraeder mit großem Fehler werden zerteilt und man erhält eine feinere Zerlegung, die als Grundlage für die wiederholte Lösung dient. Dies führt man wieder bis zur gewünschten Genauigkeit durch. Dass dieses Vorgehen bestmögliche Ergebnisse liefert ist auch Gegenstand aktueller Forschung.

0.2 Gliederung

Die Arbeit ist folgendermaßen gegliedert:

- In Abschnitt 1 stellen wir die Laplace-Gleichung in ihrer starken Formulierung vor und leiten daraus die zugehörige schwache Formulierung ab. Mit dem Begriff der Triangulierung diskretisieren wir die Gleichung schließlich und erhalten ein lineares Gleichungssystem, das die Grundlage für den numerischen Lösungsweg bildet.

- In Abschnitt 2 leiten wir Quadraturformeln für die mehrdimensionalen Integrale her, die durch den Übergang auf die schwache Formulierung auftreten. Wir verwenden dazu den Transformationssatz und den Satz von Fubini und bestimmen die eindimensionalen Integrale mittels klassischer Gauß-Quadratur. Der Abschnitt schließt mit MATLAB-Implementierungen der hergeleiteten Methoden.

- In Abschnitt 3 wird eine erste MATLAB-Implementierung zur Lösung des Laplace-Problems vorgestellt. Dem Anschein nach hängt deren Laufzeit bereits linear von der Anzahl der Elemente der Diskretisierung ab. In den darauf folgenden Seiten wird geklärt, warum hier der Schein trügt und wie sich dennoch fastlineare Abhängigkeit erreichen lässt.

- In Abschnitt 4 bedienen wir uns der Methode der Vektorisierung um die Laufzeit der Implementierung zu beschleunigen. Dazu werden kurz die hierfür entscheidenden MATLAB-Funktionen besprochen.

- Abschnitt 5 präsentiert einen wichtigen Teil adaptiver Lösungsverfahren: Netzverfeinerung und Netzvergröberung. Zuerst werden hier einige Methoden zur Bestimmung grundlegender geometrischer Beziehungen – beispielsweise die Nachbarschaftsbeziehung in einem Tetraedernetz – präsentiert. Diese werden dann für zwei Verfeinerungsschemen verwendet. Zuerst wird die sogenannte *Rotverfeinerung* vorgestellt. Diese kann nur uniform auf das gesamte Tetraedernetz angewandt werden, lässt sich dafür allerdings leicht vektorisieren. Das darauffolgende Bisektionsschema hat gänzlich andere Eigenschaften. Dieses kann dazu verwendet werden gewisse Bereiche der Triangulierung stark zu verfeinern und gleichzeitig die übrigen Teile nahezu unverändert grob zu belassen. Es lässt sich dafür aber schwer effizient in MATLAB umsetzen. Im Anhang wird deshalb eine Implementierung mithilfe der `mex`-Schnittstelle angegeben. Der Abschnitt endet mit einem zum Bisektionsverfahren gehörigen Vergröberungsverfahren.

- In Abschnitt 6 wird als letztes fehlendes Stück für das adaptive Verfahren ein Fehlerschätzer konstruiert, welcher sodann mit einigen numerischen Experimenten die Arbeit schließt.

- Der Anhang enthält die effiziente Implementierung des Bisektionsverfahrens.

0.3 Ausblick

Ein wesentlicher Teil dieser Arbeit beschäftigt sich mit der effizienten Umsetzung des Bisektionsverfahrens. Dieses erwies sich als schwerer handhabbar als das entsprechende zweidimensionale Bisektionsverfahren. Insbesondere die Frage ob es sich ähnlich dem zweidimensionalen Vorbild mittels Kantenmarkierung (siehe [6]) vektorisieren lässt, ist im Zuge dieser Arbeit ungeklärt geblieben. Somit bleiben als mögliche Erweiterungen für diese Arbeit neben der Ausdehnung auf elliptische Differentialgleichungen allgemeinerer Form noch eine genauere Analyse des Bisektionsverfahren in puncto Vektorisierbarkeit mittels Kantenmarkierung. Auch zusätzliche Fehlerschätzer bieten sich an diese Arbeit zu vertiefen.

1 Modellbeispiel & P1-Galerkin-FEM

In diesem Abschnitt wollen wir die P1-Galerkin-FEM anhand eines Modellbeispiels näher betrachten, nämlich der Laplace-Gleichung mit gemischten Dirichlet-Neumann-Randbedingungen. Zu gegebenen Funktionen $f \in L^2(\Omega)$, $u_D \in H^1(\Omega)$ und $g \in L^2(\Gamma_N)$ wollen wir eine Approximation der Lösung u folgenden Systems finden, der sogenannten starken Formulierung des Randwertproblems:

$$-\Delta u = f \text{ auf } \Omega, \tag{1.0.1}$$

$$u = u_D \text{ auf } \Gamma_D, \tag{1.0.2}$$

$$\partial_n u = g \text{ auf } \Gamma_N. \tag{1.0.3}$$

Dabei ist $\Omega \subseteq \mathbb{R}^3$ ein Lipschitz-Gebiet mit Rand Γ, der wiederum in den *Dirichlet-Rand* Γ_D und den *Neumann-Rand* Γ_N unterteilt wird. Von Γ_D und Γ_N verlangen wir, dass sie offene Teilmengen von Γ sind, sodass $\Gamma_D \cap \Gamma_N = \emptyset$ und $\Gamma = \overline{\Gamma}_D \cup \overline{\Gamma}_N$ gelte, wobei Γ_D echt positive Oberfläche $|\Gamma_D| > 0$ habe. Der Ausdruck $\partial_n u$ bezeichne die Ableitung in Richtung der äußeren Normalen n.

1.1 Schwache Formulierung und eindeutige Lösbarkeit

Da obiges Problem im Allgemeinen keine Lösung haben wird, gehen wir auf die *schwache Formulierung* von (1.0.1)–(1.0.3) über. Diese erhält man durch Multiplikation von (1.0.1) mit einer Testfunktion $v \in H^1_D(\Omega) := \{v \in H^1(\Omega) : v|_{\Gamma_D} = 0\}$ und anschließender Integration über Ω:

$$\int_\Omega f v \, d\mathbf{x} = -\int_\Omega (\Delta u) v \, d\mathbf{x} = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} - \int_\Gamma (\partial_n u) v \, ds, \tag{1.1.1}$$

wobei in der zweiten Gleichheit partielle Integration verwendet wurde. Weiters gilt wegen Eigenschaft (1.0.3) und $v|_{\Gamma_D} = 0$ die Gleichheit

$$\int_\Gamma (\partial_n u) v \, ds = \int_{\Gamma_N} (\partial_n u) v \, ds = \int_{\Gamma_N} g v \, ds. \tag{1.1.2}$$

Insgesamt suchen wir also eine Funktion $u \in H^1(\Omega)$, die (1.0.2) und

$$\int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} = \int_\Omega f v \, d\mathbf{x} + \int_{\Gamma_N} g v \, ds \quad \text{für alle } v \in H^1_D(\Omega) \tag{1.1.3}$$

erfüllt. Wir machen den Ansatz $u = u_0 + u_D$ mit unbekanntem $u_0 \in H^1_D(\Omega)$. Es folgt

$$\int_\Omega \nabla u_0 \cdot \nabla v = \int_\Omega f v \, d\mathbf{x} + \int_{\Gamma_N} g v \, ds - \int_\Omega \nabla u_D \cdot \nabla v \, d\mathbf{x} \quad \text{für } v \in H^1_D(\Omega). \tag{1.1.4}$$

Nach der Poincaré-Ungleichung definiert die linke Seite ein Skalarprodukt auf $H^1_D(\Omega)$. Ferner ist die rechte Seite ein lineares und stetiges Funktional auf $H^1_D(\Omega)$. Nach dem Satz von Riesz hat (1.1.4) deshalb eine eindeutige Lösung $u_0 \in H^1_D(\Omega)$. Nach Ansatz $u = u_0 + u_D$ hat daher auch (1.1.3) eine eindeutige Lösung $u \in H^1(\Omega)$, die (1.0.2) erfüllt. Diese ist unabhängig von der konkreten Wahl von u_D. Jede Lösung $u \in C^2(\overline{\Omega})$ von (1.0.1)–(1.0.3) erfüllt somit die schwache Formulierung. Umgekehrt ist eine Lösung $u \in C^2(\overline{\Omega})$ von (1.1.3), die zusätzlich (1.0.2) erfüllt, auch eine starke Lösung. Die Lösung von (1.1.3) ist somit der einzige sinnvolle Kandidat. (Siehe beispielsweise [10, Abschnitt 1.1])

1.2 Triangulierungen

Für die Diskretisierung des genannten Problems müssen wir zuerst einige Begrifflichkeiten einführen.

Definition 1.2.1. *Wir nennen eine Menge $T \subset \mathbb{R}^3$ einen nicht-entarteten Tetraeder, wenn es Knoten $v_1, v_2, v_3, v_4 \in \mathbb{R}^3$ gibt, sodass gilt $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ und das Volumen $|T| \neq 0$ erfüllt. Weiters seien*

$$\mathcal{N}_T := \{v_i : 1 \leq i \leq 4\} \tag{1.2.1}$$

die Menge der Knoten (nodes),

$$\mathcal{E}_T := \{\operatorname{conv}\{v_i, v_j\} : 1 \leq i < j \leq 4\} \tag{1.2.2}$$

die Menge der Kanten (edges) und

$$\mathcal{F}_T := \{\operatorname{conv}\{v_i, v_j, v_k\} : 1 \leq i < j < k \leq 4\} \tag{1.2.3}$$

die Menge der Flächen (faces) des gegebenen Tetraeders.

Definition 1.2.2. *Wir nennen eine Menge $\mathcal{T} \subseteq \mathcal{P}(\Omega)$, Teilmenge der Potenzmenge von Ω, eine Triangulierung von Ω wenn sie die folgenden Eigenschaften erfüllt:*

- *$\mathcal{T}$ ist eine endliche Menge von nicht-entarteten Tetraedern,*
- *$\overline{\Omega} = \bigcup_{T \in \mathcal{T}} T$,*
- *der Schnitt zweier Tetraeder hat Volumen $|T \cap T'| = 0$ für $T \neq T'$.*

Zu einer gegebenen Triangulierung $\mathcal{T}$ definieren wir die Mengen $\mathcal{N} := \bigcup_{T \in \mathcal{T}} \mathcal{N}_T$, $\mathcal{E} := \bigcup_{T \in \mathcal{T}} \mathcal{E}_T$ und $\mathcal{F} := \bigcup_{T \in \mathcal{T}} \mathcal{F}_T$ ihrer Knoten, Kanten und Flächen.

Dieser Triangulierungsbegriff ist für unsere Zwecke allerdings noch nicht stark genug. Wir wollen erreichen, dass sich Flächen von benachbarten Tetraedern nicht überlappen und damit sogenannte hängende Knoten vermeiden (siehe dazu Abbildung 1). Deswegen definieren wir wie folgt:

Definition 1.2.3. *Eine Triangulierung von Ω mit Rändern Γ_N und Γ_D heißt regulär wenn für je zwei Tetraeder $T, T' \in \mathcal{T}$ mit $T \neq T'$ gilt, dass der Schnitt $T \cap T'$*

- *entweder leer,*
- *oder ein gemeinsamer Knoten $v \in \mathcal{N}_{T'} \cap \mathcal{N}_T$,*
- *oder eine gemeinsame Kante $E \in \mathcal{E}_{T'} \cap \mathcal{E}_T$,*
- *oder eine gemeinsame Fläche $F \in \mathcal{F}_{T'} \cap \mathcal{F}_T$ ist,*

und zusätzlich die Triangulierung an die Ränder Γ_D und Γ_N angepasst ist. Damit meinen wir, dass für jede Fläche $F \in \mathcal{F}$ mindestens eine der Schnittflächen $F \cap \Gamma_D$ bzw. $F \cap \Gamma_N$ Oberfläche Null hat, dass also gilt:

$$\min(|F \cap \Gamma_D|, |F \cap \Gamma_N|) = 0 \quad \text{für alle } F \in \mathcal{F}. \tag{1.2.4}$$

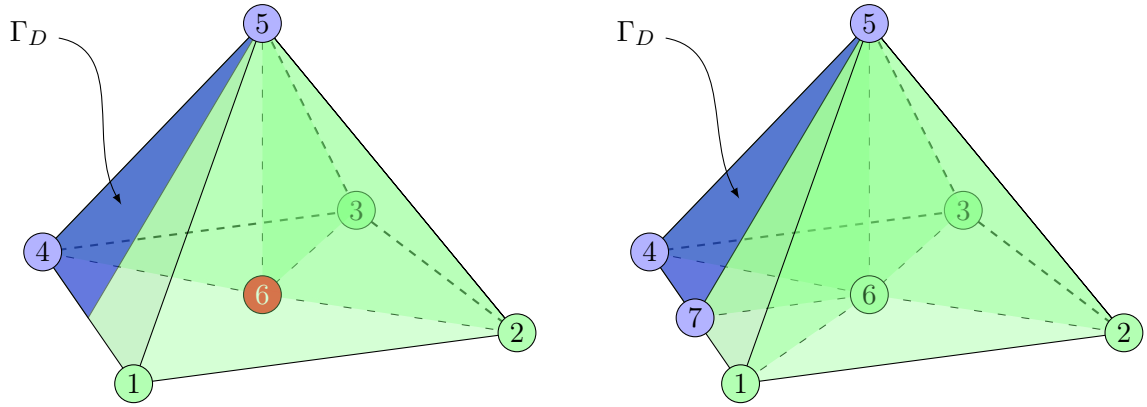


Abbildung 1: Zwei Triangulierungen einer Pyramide $P = \text{conv}\{v_1, v_2, v_3, v_4, v_5\}$, eine nichtreguläre aus drei Tetraedern und eine reguläre aus fünf Tetraedern. Die blauen Flächen stellen jeweils den Dirichlet-Rand Γ_D dar. Die Neumann-Ränder Γ_N sind dementsprechend die übrigen grünen Randflächen. Die linke Triangulierung ist aus zwei Gründen nichtregulär: Zum einen ist der Knoten v_6 ein hängender Knoten. Damit überlappt der vordere Tetraeder die beiden Hinteren, was der Definition der Regularität widerspricht. Zum anderen gilt für die linke Randfläche $F = \text{conv}\{v_1, v_4, v_5\}$ **nicht**, dass $|F \cap \Gamma_N| = 0$ oder $|F \cap \Gamma_D| = 0$ ist. Die rechte Triangulierung genügt den Voraussetzungen und ist damit regulär.

1.3 P1-Galerkin-FEM

Mit diesen Definitionen ausgerüstet, können wir nun das Problem diskretisieren. Die Galerkin-Methode besteht nun darin, (1.1.4) in einem geeigneten, endlichdimensionalen Teilraum von $H^1(\Omega)$ zu approximieren. Es sei $\mathcal{T}$ eine reguläre Triangulierung von Ω. Dann definieren wir

$$S^1(\mathcal{T}) := \{V \in C(\Omega) : V|_T \text{ ist affin für alle } T \in \mathcal{T}\} \tag{1.3.1}$$

als die Menge aller global stetigen Funktionen, die auf $\mathcal{T}$ elementweise affin sind. Seien nun die Knoten nummeriert durch $\mathcal{N} = \{v_1, \dots, v_N\}$. Als Basis $\mathcal{B}$ des Vektorraumes $S_1(\mathcal{T})$ können wir dann die Menge jener Ansatzfunktionen $\mathcal{B} = \{V_1, \dots, V_N\} \subseteq S^1(\mathcal{T})$ wählen, die eindeutig durch $V_\ell(v_k) = \delta_{kl}$ bestimmt sind. Wir werden diese Funktionen analog zum zweidimensionalen Fall mit *Hutfunktionen* bezeichnen. (Die Vorstellung eines vierdimensionalen Hutes ist dabei allerdings nicht vonnöten.) Ferner definieren wir

$$S^1_D(\mathcal{T}) := S^1(\mathcal{T}) \cap H^1_D(\Omega) = \{V \in S^1(\mathcal{T}) : V(v_\ell) = 0 \text{ für alle } v_\ell \in \mathcal{N} \cap \overline{\Gamma}_D\}. \tag{1.3.2}$$

Wenn wir die Dirichlet-Daten $u_D \in H^1(\Omega)$ als stetig auf Γ_D annehmen und die Knoten so nummerieren, dass $\mathcal{N} \cap \overline{\Gamma}_D = \{v_{n+1}, \dots, v_N\}$ gilt, dann können wir $u_D|_{\Gamma_D}$ durch den Interpolanten

$$U_D := \sum_{\ell=n+1}^{N} u_D(v_\ell) V_\ell \in S^1(\mathcal{T}) \tag{1.3.3}$$

annähern. Die diskrete Version von (1.1.3) lautet dann

$$\int_\Omega \nabla U_0 \cdot \nabla V dx = \int_\Omega fV dx + \int_{\Gamma_N} gV ds - \int_\Omega \nabla U_D \cdot \nabla V dx \quad \text{für alle } V \in S^1_D(\mathcal{T}). \tag{1.3.4}$$

Da $S^1_D(\mathcal{T})$ als endlichdimensionaler Teilraum von $H^1_D(\Omega)$ wieder ein Hilbert-Raum ist, hat (1.3.4) mit den gleichen Argumenten wie zuvor eine eindeutige Lösung $U_0 \in S^1_D(\mathcal{T})$. Diese liefert die

Näherungslösung $U := U_0 + U_D \in S^1(\mathcal{T})$ zu $u \in H^1(\Omega)$. Unser Ziel ist also die Berechnung des Koeffizientenvektors $\mathbf{x} \in \mathbb{R}^N$ der die Darstellung der Approximation

$$U = U_0 + U_D = \sum_{j=1}^{n} \mathbf{x}_j V_j + \sum_{j=n+1}^{N} u_D(v_j) V_j = \sum_{j=1}^{N} \mathbf{x}_j V_j \text{ mit } \mathbf{x}_j := u_D(v_j) \text{ für } j = n+1, \ldots, N \tag{1.3.5}$$

bezüglich unserer Basis liefert. Die diskrete Lösung U kann nun berechnet werden, indem wir die diskrete schwache Form (1.3.4) in folgendes lineares Gleichungssystem umschreiben:

$$\sum_{k=1}^{n} \mathbf{A}_{jk} \mathbf{x}_k = \mathbf{b}_j := \int_\Omega f V_j dx + \int_{\Gamma_N} g V_j ds - \sum_{k=n+1}^{N} \mathbf{A}_{jk} \mathbf{x}_k \text{ für alle } j = 1, \ldots, n. \tag{1.3.6}$$

Die Einträge der sogenannten *Steifigkeitsmatrix* $\mathbf{A}$ berechnen sich dabei durch

$$\mathbf{A}_{jk} = \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \sum_{T \in \mathcal{T}} \int_T \nabla V_j \cdot \nabla V_k \, dx \qquad \text{für alle } j, k = 1, \ldots, N. \tag{1.3.7}$$

Bevor wir nun besprechen, wie wir die Steifigkeitsmatrix $\mathbf{A} \in \mathbb{R}^{N \times N}_{\text{sym}}$ in MATLAB aufbauen um damit (1.3.6) zu lösen, werden wir uns zuerst Methoden schaffen um obige Integrale numerisch auszuwerten.

2 Quadraturregeln

In diesem Abschnitt werden wir genauer auf die Quadratur der Funktionen über den einzelnen Tetraedern bzw. den Dreiecken am Rand des Gebietes eingehen. Wir werden dazu einige wichtige Sätze aus der Analysis bemühen.

2.1 Integration über Tetraeder

Der wohl bedeutendste Satz zur Bestimmung von Integralen über allgemeinen offenen Teilmengen des $\mathbb{R}^d$ ist der Transformationssatz. Wir rufen uns deshalb diesen, sowie den Umkehrsatz, der ein Kriterium für die Differenzierbarkeit der Umkehrabbildung liefert, in Erinnerung. Die folgenden Aussagen findet man etwa in [8, S. 290 ff.].

Definition 2.1.1. *Seien U,$V \subseteq \mathbb{R}^d$ zwei offene Mengen. Eine Funktion $\Phi : U \to V$ heißt Diffeomorphismus, falls Φ eine bijektive Abbildung zwischen U und V ist, und Φ sowie Φ^{-1} stetig differenzierbar sind.*

Satz 2.1.2 (Transformationssatz, siehe z.B. [8, S. 299])**.** *Sei $\Phi : U \to V$ ein Diffeomorphismus zwischen den offenen Mengen $U, V \subseteq \mathbb{R}^d$. Eine Funktion $f : V \to \mathbb{C}$ ist genau dann integrierbar, wenn $(f \circ \Phi) \cdot |\det D\Phi| : U \to \mathbb{C}$ es ist. In diesem Fall gilt:*

$$\int_V f \, d\mathbf{x} = \int_U |\det D\Phi| \cdot f \circ \Phi \, d\mathbf{x}. \tag{2.1.1}$$

Satz 2.1.3 (Umkehrsatz, siehe z.B. [8, S. 104])**.** *Sei $\Phi : U \subseteq \mathbb{R}^d \to \mathbb{R}^d$ injektiv und stetig differenzierbar auf der offenen Menge U. Weiters gelte $\det D\Phi(\mathbf{x}) \neq 0$ für alle $\mathbf{x} \in U$. Dann ist $\Phi^{-1} : \Phi(U) \to U$ stetig differenzierbar, wobei $D(\Phi^{-1})(\Phi(\mathbf{x})) = (D\Phi(\mathbf{x}))^{-1}$ gilt.*

Im Folgenden werden wir zwei aus der Literatur entnommene Transformationen untersuchen, die wir zur Herleitung einer Quadraturformel auf Tetraedern benötigen. Die Erste ist die kanonische Erweiterung der als *Duffy-Transformation* bezeichneten Funktion auf den $\mathbb{R}^3$.

Lemma 2.1.4. *Seien* $T_{ref} := (\mathrm{conv}\{0, e_1, e_2, e_3\})^\circ = \{(x,y,z) \in \mathbb{R}^3 : x+y+z < 1 \wedge 0 < x,y,z\} = \{(x,y,z) \in \mathbb{R}^3 : 0 < x,y,z \wedge \|(x,y,z)\|_1 < 1\}$ *der offene Einheitstetraeder und* $Q_{ref} := (0,1)^3 \subseteq \mathbb{R}^3$ *der offene Einheitswürfel. Ferner seien* Φ_1 *und* Φ_2 *wie folgt definiert:*

$$\Phi_1 : \begin{cases} Q_{ref} & \to & T_{ref} \\ (x,y,z) & \mapsto & (x, (1-x)y, (1-x)(1-y)z) \end{cases} \tag{2.1.2}$$

$$\Phi_2 : \begin{cases} Q_{ref} & \to & T_{ref} \\ (x,y,z) & \mapsto & (xyz, xy(1-z), x(1-y)). \end{cases} \tag{2.1.3}$$

Dann sind Φ_1 *und* Φ_2 *zwei Diffeomorphismen zwischen* Q_{ref} *und* T_{ref} *mit den Funktionaldeterminanten*

$$\det(D\Phi_1) = (1-x)^2(1-y) \qquad \textit{und} \qquad \det(D\Phi_2) = -x^2 y. \tag{2.1.4}$$

Beweis. Zuerst bemerken wir, dass tatsächlich $\Phi_1(Q_{\mathrm{ref}}), \Phi_2(Q_{\mathrm{ref}}) \subseteq T_{\mathrm{ref}}$ gilt. Offensichtlich gilt $0 < \tilde{x}, \tilde{y}, \tilde{z}$ für alle $(\tilde{x}, \tilde{y}, \tilde{z}) \in \Phi_i(Q_{\mathrm{ref}})$, $i = 1, 2$. Schließlich gilt sowohl $\|\Phi_1((x,y,z))\|_1 = x + (1-x)y + (1-x)(1-y)z = 1 - (1-x)(1-y)(1-z) < 1$, als auch $\|\Phi_2((x,y,z))\|_1 = xyz + xy(1-z) + x(1-y) = x < 1$ für alle $(x,y,z) \in Q_{\mathrm{ref}}$, also $\Phi_1(Q_{\mathrm{ref}}), \Phi_2(Q_{\mathrm{ref}}) \subseteq T_{\mathrm{ref}}$. Nun zeigen wir die Bijektivität beider Abbildungen. Dazu prüft man, dass die beiden Abbildungen

$$g_1 : \begin{cases} T_{\mathrm{ref}} & \to & Q_{\mathrm{ref}} \\ (\tilde{x}, \tilde{y}, \tilde{z}) & \mapsto & \left(\tilde{x}, \frac{\tilde{y}}{1-\tilde{x}}, \frac{\tilde{z}}{1-\tilde{x}-\tilde{y}}\right) \end{cases}$$

$$g_2 : \begin{cases} T_{\mathrm{ref}} & \to & Q_{\mathrm{ref}} \\ (\tilde{x}, \tilde{y}, \tilde{z}) & \mapsto & \left(\tilde{x}+\tilde{y}+\tilde{z}, \frac{\tilde{x}+\tilde{y}}{\tilde{x}+\tilde{y}+\tilde{z}}, \frac{\tilde{x}}{\tilde{x}+\tilde{y}}\right). \end{cases}$$

die entsprechenden Umkehrabbildungen zu den Funktionen Φ_1 und Φ_2 sind. Zuerst zeigen wir wieder, dass auch tatsächlich $g_1(T_{\mathrm{ref}}), g_2(T_{\mathrm{ref}}) \subseteq Q_{\mathrm{ref}}$ gilt. Aus $\tilde{x}+\tilde{y}+\tilde{z} < 1$ und $\tilde{x}, \tilde{y}, \tilde{z} > 0$ für alle $(\tilde{x}, \tilde{y}, \tilde{z}) \in T_{\mathrm{ref}}$ folgt sofort, dass $g_2(T_{\mathrm{ref}}) \subseteq Q_{\mathrm{ref}}$ ist. Wir untersuchen nun die Funktion g_1. Hier gilt mithilfe obiger Voraussetzungen $0 < \tilde{x} < \tilde{x}+\tilde{y}+\tilde{z} < 1$, sowie $0 < \frac{\tilde{y}}{1-\tilde{x}} < \frac{\tilde{y}}{\tilde{y}+\tilde{z}} < 1$ und $0 < \frac{\tilde{z}}{1-\tilde{x}-\tilde{y}} < \frac{\tilde{z}}{\tilde{z}} = 1$, also auch $g_2(T_{\mathrm{ref}}) \subseteq Q_{\mathrm{ref}}$. Nun überprüfen wir, dass $g_1 \circ \Phi_1 = \mathrm{id}_{Q_{\mathrm{ref}}} = g_2 \circ \Phi_2$ und $\Phi_1 \circ g_1 = \mathrm{id}_{T_{\mathrm{ref}}} = \Phi_2 \circ g_2$. Nach Definition gilt für Φ_1:

$$\begin{aligned} g_1 \circ \Phi_1(x,y,z) &= g_1(x, (1-x)y, (1-x)(1-y)z) \\ &= \left(x, \frac{(1-x)y}{1-x}, \frac{(1-x)(1-y)z}{1-x-(1-x)y}\right) \\ &= (x,y,z) \end{aligned}$$

$$\begin{aligned} \Phi_1 \circ g_1(\tilde{x}, \tilde{y}, \tilde{z}) &= \Phi_1\left(\tilde{x}, \frac{\tilde{y}}{1-\tilde{x}}, \frac{\tilde{z}}{1-\tilde{x}-\tilde{y}}\right) \\ &= \left(\tilde{x}, (1-\tilde{x})\frac{\tilde{y}}{1-\tilde{x}}, (1-\tilde{x})(1-\frac{\tilde{y}}{1-\tilde{x}})\frac{\tilde{z}}{1-\tilde{x}-\tilde{y}}\right) \\ &= \left(\tilde{x}, \tilde{y}, \frac{(1-\tilde{x})(1-\tilde{x}-\tilde{y})\tilde{z}}{(1-\tilde{x})(1-\tilde{x}-\tilde{y})}\right) \\ &= (\tilde{x}, \tilde{y}, \tilde{z}). \end{aligned}$$

Genauso gehen wir für die Funktion Φ_2 vor:

$$\begin{aligned} g_2 \circ \Phi_2(x,y,z) &= g_2(xyz, xy(1-z), x(1-y)) \\ &= \left(xyz + xy(1-z) + x(1-y), \frac{xyz + xy(1-z)}{xyz + xy(1-z) + x(1-y)}, \frac{xyz}{xyz + xy(1-z)} \right) \\ &= \left(x, \frac{xy}{x}, \frac{xyz}{xy} \right) \\ &= (x,y,z) \end{aligned}$$

$$\begin{aligned} \Phi_2 \circ g_2(\tilde{x},\tilde{y},\tilde{z}) &= \Phi_2 \left(\tilde{x} + \tilde{y} + \tilde{z}, \frac{\tilde{x}+\tilde{y}}{\tilde{x}+\tilde{y}+\tilde{z}}, \frac{\tilde{x}}{\tilde{x}+\tilde{y}} \right) \\ &= \left(\tilde{x}, (\tilde{x}+\tilde{y}+\tilde{z}) \frac{\tilde{x}+\tilde{y}}{\tilde{x}+\tilde{y}+\tilde{z}} \left(1 - \frac{\tilde{x}}{\tilde{x}+\tilde{y}}\right), (\tilde{x}+\tilde{y}+\tilde{z}) \left(1 - \frac{\tilde{x}+\tilde{y}}{\tilde{x}+\tilde{y}+\tilde{z}}\right) \right) \\ &= \left(\tilde{x}, (\tilde{x}+\tilde{y}) \frac{\tilde{y}}{\tilde{x}+\tilde{y}}, (\tilde{x}+\tilde{y}+\tilde{z}) \frac{\tilde{z}}{\tilde{x}+\tilde{y}+\tilde{z}} \right) \\ &= (\tilde{x},\tilde{y},\tilde{z}). \end{aligned}$$

Also gilt $\Phi_1^{-1} = g_1$ und $\Phi_2^{-1} = g_2$ und damit die Bijektivität von Φ_1 und Φ_2. Die Abbildungen Φ_1 und Φ_2 sind stetig differenzierbar mit den Jacobi-Matrizen

$$D\Phi_1((x,y,z)) = \begin{pmatrix} 1 & 0 & 0 \\ -y & 1-x & 0 \\ -(1-y)z & -(1-x)z & (1-x)(1-y) \end{pmatrix}$$

und

$$D\Phi_2((x,y,z)) = \begin{pmatrix} yz & xz & xy \\ y(1-z) & x(1-z) & -xy \\ 1-y & -x & 0 \end{pmatrix}.$$

Die Dreiecksgestalt von $D\Phi_1$ liefert $\det(D\Phi_1(x,y,z)) = (1-x)^2(1-y)$. Mit elementaren Zeilenumformungen erhalten wir $\det(D\Phi_2(x,y,z)) = -x^2y$:

$$\begin{aligned} \det D\Phi_2 &= \det \left(\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot D\Phi_2 \right) \\ &= \det \begin{pmatrix} 1 & 0 & 0 \\ y(1-z) & x(1-z) & -xy \\ 1-y & -x & 0 \end{pmatrix} \\ &= -\det \begin{pmatrix} 1 & 0 & 0 \\ 1-y & -x & 0 \\ y(1-z) & x(1-z) & -xy \end{pmatrix} \\ &= -x^2y. \end{aligned}$$

Da beide Determinanten für $(x,y,z) \in Q_{\text{ref}}$ ungleich null sind, erhalten wir mit dem Umkehrsatz (2.1.3), dass die Inversen auch stetig differenzierbar sind und somit Φ_1 sowie Φ_2 zwei Diffeomorphismen von Q_{ref} nach T_{ref}. □

Die beiden Transformationen aus Lemma 2.1.4 werden in Abbildung 2 visualisiert. Wir werden etwas später noch darauf eingehen, dass die Funktionen sichtlich zu unterschiedlichen Punkten hin verdichten. Zuvor benötigen wir allerdings noch eine Formel für Integrale über allgemeine

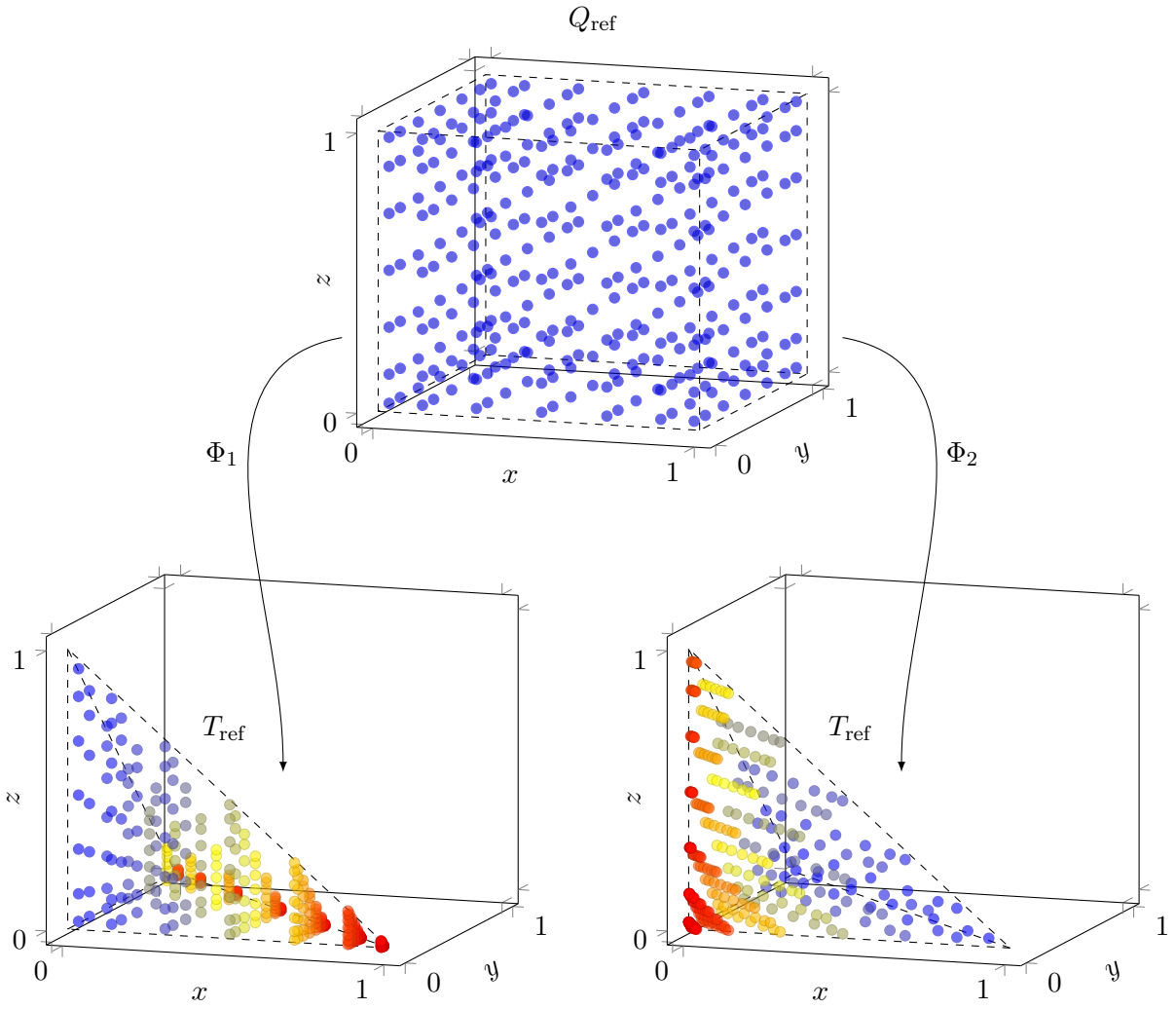


Abbildung 2: Gauß-Knoten des Referenzwürfels unter beiden Transformationen. Die Transformation Φ_1 verdichtet zum Punkt $(1,0,0)$ hin, sowie zur Kante $\text{conv}\{(1,0,0),(0,1,0)\}$, wohingegen Φ_2 zu $(0,0,0)$ und der Kante $\text{conv}\{(0,0,0),(0,0,1)\}$ verdichtet.

Tetraeder. Diese lassen sich aber als affine Transformationen unseres Referenztetraeders darstellen.

Bemerkung 2.1.5. *Jede affine Transformation* $\Phi_{affin} : U \subseteq \mathbb{R}^3 \to \mathbb{R}^3$, $\Phi_{affin}(\mathbf{x}) := A\mathbf{x} + b$, $A \in \mathbb{R}^{3\times 3}$, $b \in \mathbb{R}^3$, *mit* U *offen, ist im Falle einer regulären Matrix* A *ein Diffeomorphismus auf sein Bild. Für die Funktionaldeterminante gilt* $\det(D\Phi_{affin}) = \det A$.

Damit können wir nun Integrale über beliebige Tetraeder auf Integrale über den Referenzwürfel zurückführen.

Korollar 2.1.6. *Sei* $T = (\text{conv}\{v_1, v_2, v_3, v_4\})^\circ \subseteq \mathbb{R}^3$ *ein beliebiger Tetraeder mit den Eckpunkten* $\{v_1, v_2, v_3, v_4\}$. *Sei weiters* Φ_i *eine der beiden Transformationen aus Lemma 2.1.4. Wir definieren die Transformation* $\Phi_T(\mathbf{x}) := A\mathbf{x} + v_4$ *mit der Matrix* $A := (v_1 - v_4, v_2 - v_4, v_3 - v_4) \in \mathbb{R}^{3\times 3}$

von T auf T_{ref}. Dann gilt:

$$\int_T f\,d\mathbf{x} = |\det A| \cdot \int_{Q_{ref}} |\det D\Phi_i| \cdot f \circ \Phi_T \circ \Phi_i\,d\mathbf{x}. \tag{2.1.5}$$

Beweis. Erster Fall: $\det A \neq 0$. Man prüft leicht, dass $\Phi_T(T_{\text{ref}}) = T$. Da nun Q_{ref} und T offene Mengen sind und A regulär ist, stellt $\Phi := \Phi_{\text{T}} \circ \Phi_i$ als Verknüpfung zweier Diffeomorphismen einen Diffeomorphismus von Q_{ref} auf T dar. Wir können Satz 2.1.2 mit $U := Q_{\text{ref}}$, $V := T$, $\Phi := \Phi_{\text{T}} \circ \Phi_i$ anwenden. Es folgt somit

$$\begin{aligned}
\int_T f\,d\mathbf{x} &= \int_{Q_{\text{ref}}} |\det D\Phi| \cdot f \circ \Phi\,d\mathbf{x} \\
&= \int_{Q_{\text{ref}}} |\det(D\Phi_T)(\Phi_i(x,y,z)) \cdot \det D\Phi_i(x,y,z)| \cdot f \circ \Phi_{\text{T}} \circ \Phi_i(x,y,z)\,d(x,y,z) \\
&= \int_{Q_{\text{ref}}} |\det A| \cdot |\det D\Phi_i(x,y,z)| \cdot f \circ \Phi_{\text{T}} \circ \Phi_i(x,y,z)\,d(x,y,z) \\
&= |\det A| \cdot \int_{Q_{\text{ref}}} |\det D\Phi_i| \cdot f \circ \Phi_{\text{T}} \circ \Phi_i\,d\mathbf{x}
\end{aligned}$$

Zweiter Fall: $\det A = 0$. In diesem Fall kann der Transformationssatz nicht angewendet werden. Jedoch folgt aus $\det A = 0$, dass $T = \Phi_T(T_{\text{ref}})$ ein entarteter Tetraeder und damit eine λ_3-Nullmenge ist. Somit sind beide Seiten der Gleichung null und (2.1.5) gilt trivialerweise. □

2.2 Gauß'sche Quadraturformeln

Um die in Korollar 2.1.6 erhaltene Integralformel numerisch auszuwerten, werden wir die Methode der klassischen Gauß-Quadratur verwenden. Diese integriert mit n Auswertungspunkten, Polynome bis zum Grad $2n-1$ exakt. Die Güte der Approximationen für nichtpolynomielle Funktionen werden wir anhand einiger Beispiele prüfen.

Definition 2.2.1. *Wir definieren für zwei Polynome p und q das Skalarprodukt $\langle p, q\rangle := \int_0^1 p(x)q(x)dx$. Die Polynome $(p_n)_{n\in\mathbb{N}}$, die durch Gram-Schmidt-Orthogonalisierung der Monome bezüglich diesem Skalarprodukt entstehen, nennen wir monische Orthogonalpolynome.*

Satz 2.2.2 (Vgl. z.B. [11, Kapitel 6])**.** *Seien die Punkte $x_i^{(n)} \in \mathbb{R}$, $i = 1, \ldots, n$ die n Nullstellen des monischen Orthogonalpolynoms p_n. Wir definieren*

$$L_i(x) := \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{x - x_j^{(n)}}{x_i^{(n)} - x_j^{(n)}}, \qquad \omega_i^{(n)} := \int_0^1 L_i(x)dx. \tag{2.2.1}$$

Dann gilt: Die Quadraturformel

$$Q_n(f) := \sum_{i=1}^{n} \omega_i^{(n)} f\big(x_i^{(n)}\big) \tag{2.2.2}$$

erfüllt

$$\int_0^1 f dx = Q_n(f) \qquad \text{für alle } f \in \mathcal{P}_{2n-1}. \tag{2.2.3}$$

Diese Quadraturformel wollen wir nun auch auf Funktionen $f : \mathbb{R}^3 \to \mathbb{R}$ erweitern. Zuvor benötigen wir allerdings noch einige Definitionen.

Definition 2.2.3. *Mit den Multiindexschreibweisen für die Vektoren* $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ *und* $\mathbf{k} = (k_1, \ldots, k_d) \in \mathbb{N}^d$

$$\mathbf{x}^{\mathbf{k}} := \prod_{j=1}^{d} x_j^{k_j} \qquad \textit{und} \qquad |\mathbf{k}| = \sum_{j=1}^{d} k_j, \tag{2.2.4}$$

definieren wir die Menge der d-dimensionalen Polynome vom Totalgrad kleiner oder gleich n als

$$\mathcal{P}_n(\mathbb{R}^d) := \Big\{ \sum_{|\mathbf{k}|\leq n} a_{\mathbf{k}} \mathbf{x}^{\mathbf{k}} : a_{\mathbf{k}} \in \mathbb{R} \Big\}. \tag{2.2.5}$$

Den Totalgrad eines Polynoms $p(\mathbf{x}) = \sum_{|\mathbf{k}|\leq n} a_{\mathbf{k}} \mathbf{x}^{\mathbf{k}}$ *definieren wir dann als*

$$\deg(p) := \max_{a_{\mathbf{k}} \neq 0} |\mathbf{k}|. \tag{2.2.6}$$

Wir definieren ferner den x_j*-Grad als*

$$\deg_{x_j}(p) := \max_{a_{\mathbf{k}} \neq 0} k_j. \tag{2.2.7}$$

Satz 2.2.4. *Seien* $x_i^{(n)}$ *und* $\omega_i^{(n)}$ *die klassischen Gauß-Knoten und Gauß-Gewichte aus Satz 2.2.2. Definiere die Tensorknoten und Tensorgewichte:*

$$\mathbf{x}_{i,j,k}^{\square(n)} := \big(x_i^{(n)}, x_j^{(n)}, x_k^{(n)}\big) \in \mathbb{R}^3, \qquad \omega_{i,j,k}^{\square(n)} := \omega_i^{(n)} \omega_j^{(n)} \omega_k^{(n)} \in \mathbb{R}. \tag{2.2.8}$$

Dann gilt: Die Quadraturformel

$$Q_n^{\square}(f) := \sum_{i,j,k=1}^{n} \omega_{i,j,k}^{\square(n)} \, f\big(\mathbf{x}_{i,j,k}^{\square(n)}\big) \tag{2.2.9}$$

erfüllt

$$\int_{Q_{ref}} f \, d(x,y,z) = Q_n^{\square}(f) \qquad \textit{für alle } f \in \mathcal{P}_{3(2n-1)}(\mathbb{R}^3) \textit{ mit } \max_{j\in\{1,2,3\}} \deg_{x_j}(f) \leq 2n-1. \tag{2.2.10}$$

Beweis. Sei f von obiger Form. Anwendung des Satzes von Fubini (siehe z.B. [8, S. 289]) liefert:

$$\int_{Q_{\mathrm{ref}}} f(x,y,z) \, d(x,y,z) = \int_0^1 \int_0^1 \int_0^1 f(x,y,z) \, dx \, dy \, dz$$

Für festes y und z ist ist der Integrand von $\int_0^1 f(x,y,z)\,dx$ ein Polynom in x, dessen Grad $\deg f(\,.\,,y,z) = \deg_{x_1} f \leq 2n-1$ ist. Damit können wir Satz 2.2.2 anwenden und es folgt:

$$\begin{aligned} \int_{Q_{\mathrm{ref}}} f \, d(x,y,z) &= \int_0^1 \int_0^1 \sum_{i=1}^{n} \omega_i^{(n)} f\big(x_i^{(n)}, y, z\big) \, dy \, dz \\ &= \sum_{i=1}^{n} \omega_i^{(n)} \int_0^1 \int_0^1 f\big(x_i^{(n)}, y, z\big) \, dy \, dz \end{aligned}$$

Für y und z können wir genauso argumentieren und erhalten damit, dass gilt

$$\begin{aligned}\int_{Q_{\text{ref}}} f\,d(x,y,z) &= \sum_{i=1}^{n} \omega_i^{(n)} \int_0^1 \int_0^1 f\big(x_i^{(n)}, y, z\big)\,dy\,dz \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \omega_i^{(n)}\omega_j^{(n)} \int_0^1 f\big(x_i^{(n)}, x_j^{(n)}, z\big)\,dz \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} \omega_i^{(n)}\omega_j^{(n)}\omega_k^{(n)} f\big(x_i^{(n)}, x_j^{(n)}, x_k^{(n)}\big) \\ &= \sum_{i,j,k=1}^{n} \omega_{i,j,k}^{\square(n)}\, f\big(\mathbf{x}_{i,j,k}^{\square(n)}\big).\end{aligned}$$

Unser Satz ist damit gezeigt. □

Nun haben wir das nötige Werkzeug um auch Funktionen numerisch über Tetraeder zu integrieren.

Korollar 2.2.5. *Sei $T = (\operatorname{conv}\{v_1, v_2, v_3, v_4\})^\circ \subseteq \mathbb{R}^3$ ein beliebiger Tetraeder und Φ_i eine der beiden Transformationen aus Lemma 2.1.4. Wir definieren erneut $\Phi_T(\mathbf{x}) := A\mathbf{x} + v_4$ mit der Matrix $A := (v_1 - v_4, v_2 - v_4, v_3 - v_4) \in \mathbb{R}^{3\times 3}$. Wir definieren weiters die transformierten Gauß-Knoten*

$$\mathbf{x}_{i,j,k}^{\triangle(n)} := \Phi_T \circ \Phi_i\big(\mathbf{x}_{i,j,k}^{\square(n)}\big) \tag{2.2.11}$$

und die zugehörigen Gauß-Gewichte

$$\omega_{i,j,k}^{\triangle(n)} := \big|\det A\big|\,\big|\det D\Phi_i\big(\mathbf{x}_{i,j,k}^{\square(n)}\big)\big|\,\omega_{i,j,k}^{\square(n)}. \tag{2.2.12}$$

Dann gilt: Die Quadraturformel

$$Q_{n,T}^{\triangle}(f) := \sum_{i,j,k=1}^{n} \omega_{i,j,k}^{\triangle(n)} f\big(\mathbf{x}_{i,j,k}^{\triangle(n)}\big) \tag{2.2.13}$$

erfüllt

$$\int_T f\,d\mathbf{x} = Q_{n,T}^{\triangle}(f) \qquad \text{für alle } f \in \mathcal{P}_{2n-3}\big(\mathbb{R}^3\big). \tag{2.2.14}$$

Beweis. Wir wollen Satz 2.2.4 auf die durch Korollar 2.1.6 erhaltene Funktion

$$g := |\det A| \cdot |\det D\Phi_i| \cdot f \circ \Phi_{\mathrm{T}} \circ \Phi_i$$

anwenden. Wir zeigen also, dass g die entsprechenden Voraussetzungen erfüllt. Zuerst bemerken wir, dass $\det A \in \mathbb{R}$ ist und alle Komponenten der Funktionen Φ_1 und Φ_2 Polynome $p \in \mathcal{P}_3(\mathbb{R}^3)$ mit $\max_{j\in\{1,2,3\}} \deg_{x_j}(p) \leq 1$ sind. Da Φ_T lediglich eine affine Transformation darstellt, gilt selbiges auch für $\Phi_T \circ \Phi_i$. Verknüpfung des Polynoms f mit $\Phi_T \circ \Phi_i$ liefert dann ein Polynom $f \circ \Phi_T \circ \Phi_i \in \mathcal{P}_{3(2n-3)}(\mathbb{R}^3)$ für das gilt

$$\max_{j\in\{1,2,3\}} \deg_{x_j}(f \circ \Phi_T \circ \Phi_i) \leq \deg(f) = 2n - 3.$$

Nun erfüllen die Funktionen $|\det D\Phi_1| = -\det D\Phi_1 = -(1-x)^2(1-y)$ und $|\det D\Phi_2| = -\det D\Phi_2 = x^2 y$ für $(x,y,z) \in Q_{\text{ref}}$ die Eigenschaft

$$\max_{j\in\{1,2,3\}} \deg_{x_j}(|\det D\Phi_1|) = \max_{j\in\{1,2,3\}} \deg_{x_j}(|\det D\Phi_2|) = 2.$$

Was die gewünsche Voraussetzung

$$\begin{aligned}\max_{j\in\{1,2,3\}} \deg_{x_j}(g) &= \max_{j\in\{1,2,3\}} \deg_{x_j}(|\det A| \cdot |\det D\Phi_i| \cdot f\circ\Phi_{\mathrm{T}}\circ\Phi_i)\\ &\le \max_{j\in\{1,2,3\}} \deg_{x_j}(|\det D\Phi_i|) + \max_{j\in\{1,2,3\}} \deg_{x_j}(f\circ\Phi_T\circ\Phi_i)\\ &= 2n-1\end{aligned}$$

für Satz 2.2.4 liefert. Nun ist also

$$\begin{aligned}\int_T f d\mathbf{x} &= \int_{Q_{\text{ref}}} |\det A| \cdot |\det D\Phi_i| \cdot f\circ\Phi_{\mathrm{T}}\circ\Phi_i \, d\mathbf{x}\\ &= \sum_{i,j,k=1}^{n} \omega_{i,j,k}^{\square(n)} \left(|\det A| \cdot |\det D\Phi_i| \cdot f\circ\Phi_{\mathrm{T}}\circ\Phi_i\right)\left(\mathbf{x}_{i,j,k}^{\square(n)}\right)\\ &= \sum_{i,j,k=1}^{n} \omega_{i,j,k}^{\triangle(n)} f\left(\mathbf{x}_{i,j,k}^{\triangle(n)}\right).\end{aligned}$$

Damit ist die Aussage bewiesen. □

Listing 1: Gauss-Quadratur über Tetraeder

```
function [XYZ,W,sizeT,V] = tetquad(n,coord,method)
if nargin<3 || method==1
    %*** Definition der Duffy-Transformation Φ1
    trafo = @(x,y,z) [x,(1-x).*y,(1-x).*(1-y).*z];
    trafodet = @(x,y,z) (1-x).*(1-x).*(1-y);
else
    %*** Definition der Transformation Φ2
    trafo = @(x,y,z) [x.*y.*z,x.*y.*(1-z),x.*(1-y)];
    trafodet = @(x,y,z) (x.*x.*y);
end
%*** Generierung der eindimensionalen Gauss-Knoten und -gewichte fuer [0,1]
[nodes, weights] = gauss1D(n,0,1);
%*** Erzeugung des Gauss-Knotengitters auf Q_ref
X = repmat(nodes',[n,1,n]);
Y = shiftdim(X,1);
Z = shiftdim(X,2);
%*** Bestimmung der zu den Knoten gehoerenden Gewichte
W = repmat(weights,[n,1,n]);
W = W.*shiftdim(W,1).*shiftdim(W,2);
%*** Transformation der Gauss-Knoten von Q_ref auf T_ref
XYZ = trafo(X(:),Y(:),Z(:));
%*** Hutfunktionen bzw. Baryzentrische Koordinaten
V = [XYZ,1-sum(XYZ,2)];
%*** Multiplikation der Determinante der Transformation auf die Gewichte
W = W.*abs(trafodet(X,Y,Z));
%*** Affine Transformation von T_ref auf T
A = [eye(3),-ones(3,1)]*coord;
XYZ = XYZ*A+repmat(coord(4,:),n*n*n,1);
```

```
29 %*** Anpassung der Gewichte mittels Determinante der affinen Transformation
30 W = abs(det(A))*W(:)';
31 sizeT = abs(det(A))/6;
```

Die Funktion `tetquad` aus Listing 1 realisiert nun Korollar 2.2.5. Zu einem gegebenen Tetraeder liefert es die entsprechenden Gauß-Knoten und Gewichte. Es kann dabei zwischen den beiden Diffeomorphismen aus Lemma 2.1.4 ausgewählt werden. Der Tetraeder wird als Array der Eckpunkte `coord` $\in \mathbb{R}^{4\times 3}$ übergeben. Die zugrundeliegende eindimensionale Gauß-Quadratur wird dabei wie bei $Q^{\triangle}_{n,T}$ mit n Knoten durchgeführt. `tetquad` liefert dann einen $n^3 \times 3$ Array der Gauß-Knoten `XYZ` und den dazugehörigen Array der Gauß-Gewichte `W` $\in \mathbb{R}^{1\times n^3}$. Zusätzlich können noch die baryzentrischen Koordinaten V der Gauß-Knoten zurückgegeben werden. Die Quadratur kann schließlich mit dem Befehl

```
integral = W*f(XYZ);
```

durchgeführt werden, sofern die Funktion `f` für eine $(n \times 3)$-Matrix den Spaltenvektor der n Funktionswerte zurückgibt.

2.3 Numerische Experimente

Wir wollen nun anhand einiger Beispiele die Güte des soeben entwickelten Verfahrens testen. In Abbildung 3 und Abbildung 4 wird der relative Fehler des Verfahrens in Abhängigkeit vom Grad der eindimensionalen Gauß-Quadratur abgebildet. Es wurden dabei die angegebenen Funktionen auf dem Referenztetraeder T_{ref} mit der Funktion `tetquad` integriert. Abbildung 3 bestätigt unsere Erwartung bezüglich des Exaktheitsgrades der Quadraturregel. Polynome vom Grad $2n-3$ werden durch $Q^{\triangle}_{n,T}$ – bis auf Maschinengenauigkeit – exakt integriert. In Abbildung 4 werden die Unterschiede zwischen den beiden Transformationen deutlich. Für die gewählten Funktionen ist es günstiger, zum Punkt $(0,0,0)$ hin, also mittels Φ_2 zu verdichten. Welche Transformation im konkreten Fall bessere Ergebnisse erzielt ist aber im Allgemeinen a priori nicht bekannt. Wenn wir bedenken, dass abhängig von der Reihenfolge der Knoten in `coord`, der Verdichtungspunkt zu verschiedenen Eckpunkten hin transformiert wird, dann stellt sich folgende Frage: Welche der Transformationen liefert bessere Ergebnisse, wenn wir die Verdichtungspunkte zum selben Punkt hin transformieren? Um also einen objektiveren Vergleich der beiden Transformationen zu erhalten, wurden auch Tests mit geänderter Knotenreihenfolge durchgeführt, in der Form, dass beide Verfahren zum selben Knoten hin verdichten. (Im Speziellen wurde `tetquad(n,coord,1)` mit `tetquad(n,coord(4:-1:1,:),2)` verglichen). Das erstaunliche Ergebnis dieses Vergleiches sind exakt gleiche Approximationen bei beiden Verfahren. Genauere Analyse der Funktionen hat folgende Begründung dafür gebracht.

Bemerkung 2.3.1. *Die Abbildung* $\Phi_1(x,y,z) := (x,(1-x)y,(1-x)(1-y)z)$ *kann durch affine Transformation übergeführt werden in* $((1-x)(1-y)(1-z),(1-x)(1-y)z,(1-x)y)$*, das entspricht aber genau* $\Phi_2(1-x,1-y,1-z)$*.*

$$\begin{aligned}
\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ (1-x)y \\ (1-x)(1-y)z \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} &= \begin{pmatrix} 1-x-(1-x)y-(1-x)(1-y)z \\ (1-x)(1-y)z \\ (1-x)y \end{pmatrix} \\
&= \begin{pmatrix} (1-x)(1-y)(1-z) \\ (1-x)(1-y)z \\ (1-x)y \end{pmatrix} \\
&= \Phi_2 \begin{pmatrix} 1-x \\ 1-y \\ 1-z \end{pmatrix}
\end{aligned}$$

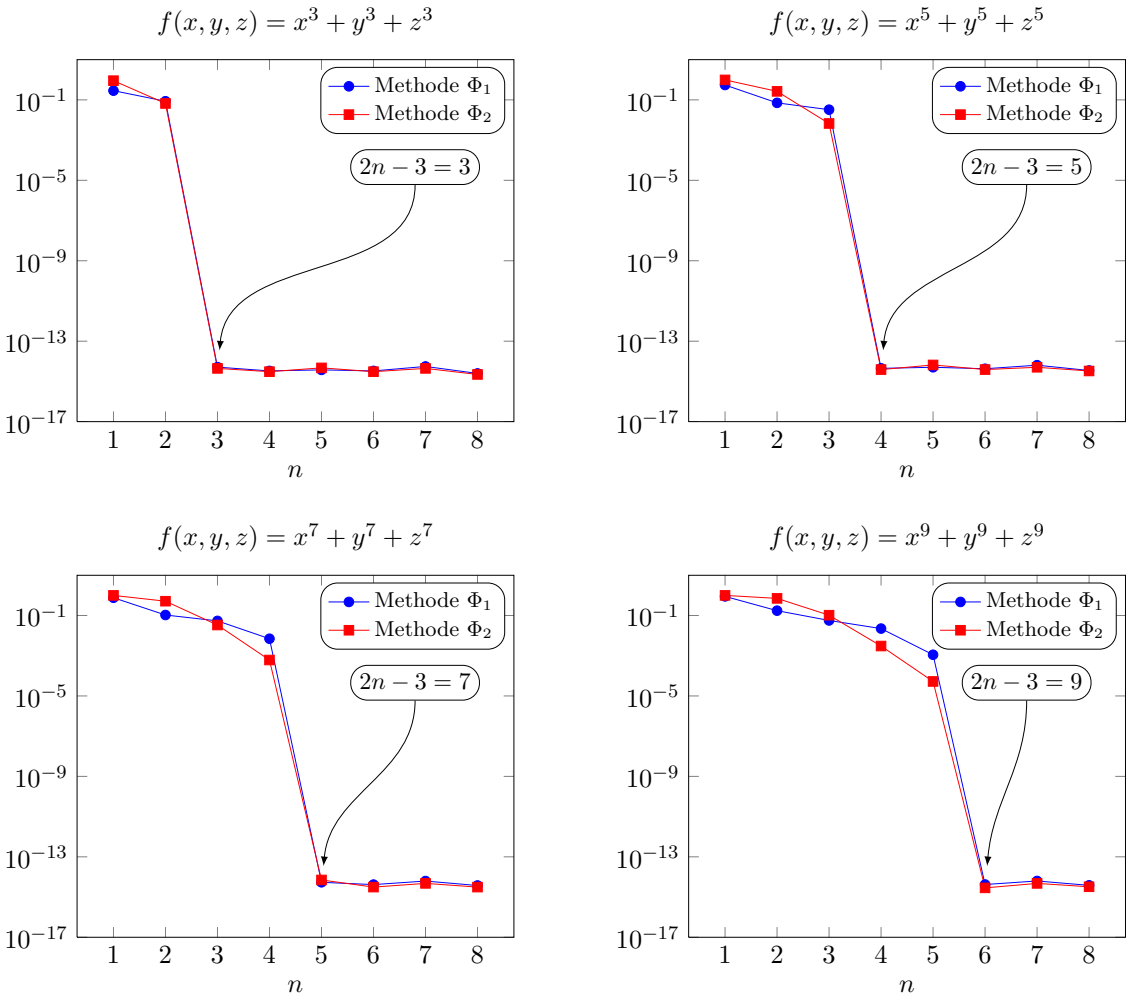


Abbildung 3: Relative Fehler der Quadraturformeln in Abhängigkeit der Anzahl n der Gauß-Knoten je Dimension. Gut erkennbar ist die – bis auf Maschinengenauigkeit – exakte Approximation der Polynome vom Grad $2n-3$.

Da die Gauß-Knoten des Intervalls $[0,1]$ *symmetrisch um den Mittelpunkt* 0.5 *angeordnet sind bedeutet das, dass beide Verfahren dieselben Gauß-Knoten liefern, falls die Verdichtungspunkte entsprechend gewählt werden. Für unsere Zwecke sind deshalb die beiden Transformationen als gleichwertig anzusehen.*

Schließlich sei mit Listing 2 noch die Implementierung der Gauß-Quadratur über Dreiecksflächen angeführt.

Listing 2: Gauss-Quadratur über Dreiecke

```
1 function [XYZ,W,sizeT,V] = triquad(n,coord)
2 if isempty(coord), coord = [eye(2);zeros(1,2)]; end
3 %*** Definition der Duffy-Transformation
4 trafo = @(x,y) [x,(1-x).*y];
5 trafodet = @(x,y) (1-x);
6 %*** Generierung der eindimensionalen Gauss-Knoten und -gewichte fuer [0,1]
7 [nodes, weights] = gauss1D(n,0,1);
8 %*** Erzeugung des Gauss-Knotengitters auf [0,1]^2
```

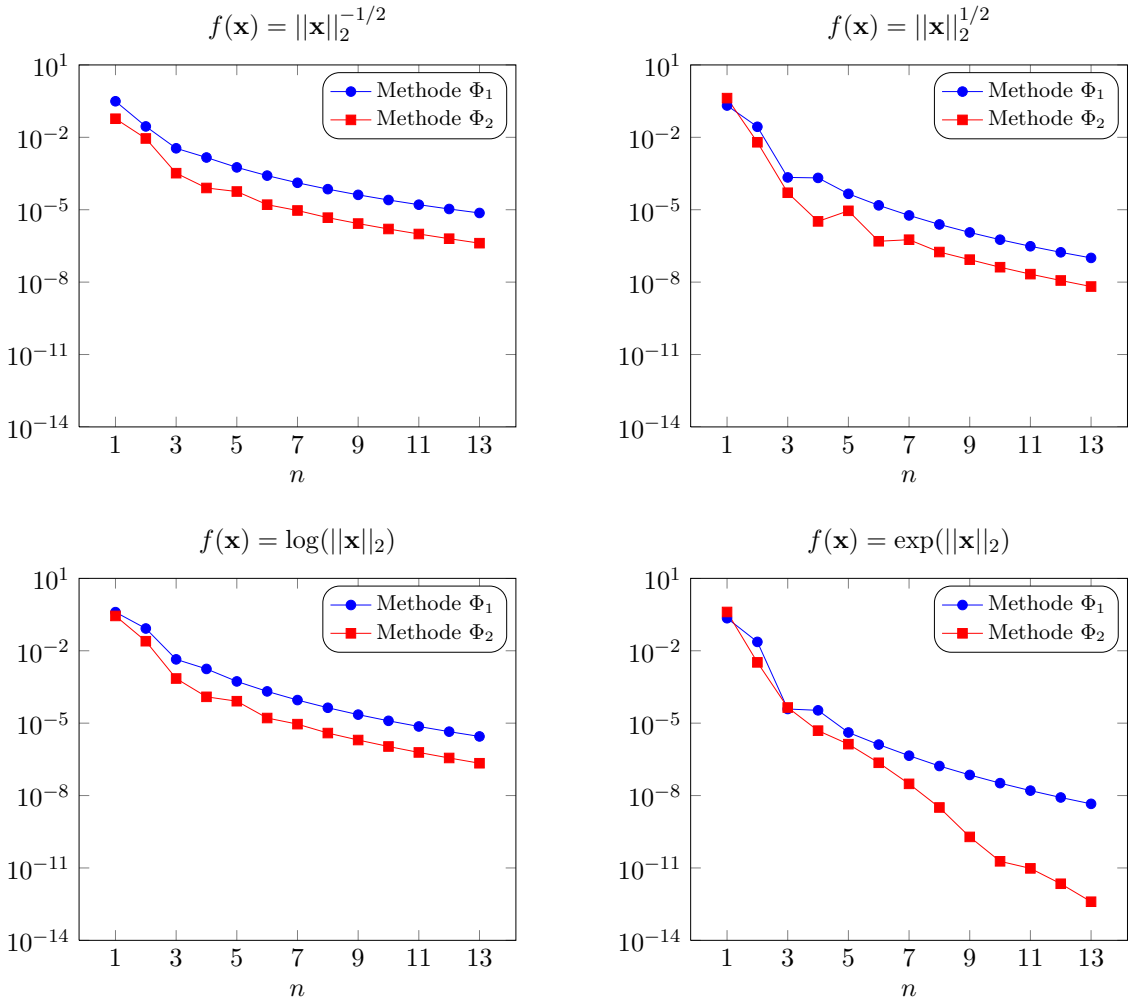


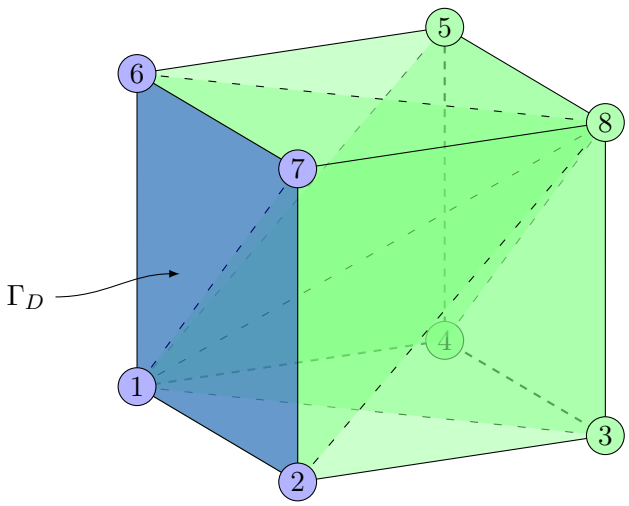
Abbildung 4: Relative Fehler der Quadraturformeln in Abhängigkeit der Anzahl n der Gauß-Knoten je Dimension. Verwendung der Transformation Φ_2 liefert bessere Ergebnisse, denn bei den verwendeten Funktionen ist es günstig hin zum Punkt $(0, 0, 0)$ zu verdichten. Die Funktionen $||\mathbf{x}||_2^{-1/2}$ und $\log(||\mathbf{x}||_2)$ haben dort Polstellen.

```
X = repmat(nodes',[n,1]);
Y = X';
%*** Bestimmung der zu den Knoten gehoerenden Gewichte
W = repmat(weights,[n,1]);
W = W.*W';
%*** Anwendung der Transformation auf die Gauss-Knoten
XY = trafo(X(:),Y(:));
%*** Hutfunktionen bzw. Baryzentrische Koordinaten
V = [XY,1-sum(XY,2)];
%*** Multiplikation der Determinante der Transformation auf die Gewichte
W = W.*abs(trafodet(X,Y));
%*** Affine Transformation vom Referenzdreieck auf coord
A = [eye(2),-ones(2,1)]*coord;
XYZ = XYZ*A+repmat(coord(3,:),n*n,1);
%*** Anpassung der Gewichte an das Oberflaechenmass
W = sqrt(det(A*A'))*W(:)';
sizeT = sqrt(det(A'*A))/2;
```

3 MATLAB-Implementierung der P1-Galerkin-FEM

Dieser Abschnitt ist der Implementierung der Galerkin-Methode gewidmet. Zentral dafür sind die beiden Gleichungen (1.3.6) und (1.3.7) aus dem Abschnitt 1. Wir werden zuerst eine Datenstruktur zur Speicherung der Triangulierung besprechen, mit der wir in weiterer Folge direkt eben erwähnte Gleichungen implementieren. Wir werden auch klären, warum eine allzu naive Implementierung quadratischen Aufwand liefert.

3.1 Datenstruktur in MATLAB



	coordinates		
1	0	0	0
2	0	1	0
3	1	1	0
4	1	0	0
5	1	0	1
6	0	0	1
7	0	1	1
8	1	1	1

	elements			
1	1	2	7	8
2	1	2	3	8
3	1	4	3	8
4	1	4	5	8
5	1	6	5	8
6	1	6	7	8

	dirichlet		
1	1	2	7
2	1	6	7

	neumann		
1	2	7	8
2	2	3	8
3	3	4	8
4	4	5	8
5	1	4	5
6	1	5	6
7	1	2	3
8	1	3	4
9	5	6	8
10	6	7	8

Abbildung 5: Reguläre Triangulierung des Würfels $\Omega = [0,1]^3$ aus 6 Tetraedern. Die Einträge in `elements` beziehen sich auf die Zeilen von `coordinates`. Anhand von `elements` ist gut erkennbar, dass alle Tetraeder die Kante $\mathrm{conv}\{v_1, v_8\}$ gemeinsam haben. Der Dirichlet-Rand Γ_D, dargestellt durch die Matrix `dirichlet`, ist blau gekennzeichnet. Die Knoten in $\mathcal{N} \cap \Gamma_D$ sind blau hinterlegt, die freien Knoten grün. Der Neumann-Rand wird durch `neumann` angegeben und besteht aus den übrigen 10 Randflächen.

Für die Darstellung der Triangulierung in MATLAB verwenden wir das in MATLAB gebräuchliche Simplex-Vertex-Format (siehe auch [1, Abschnitt 4]). Eine Beispieltriangulierung, sowie die entsprechenden Datenstrukturen sind Abbildung 5 zu entnehmen. Um genanntes Format zu erhalten, ordnen wir jeder regulären Triangulierung $\mathcal{T}$ mit Rändern Γ_D und Γ_N die Arrays `elements`, `coordinates`, `dirichlet` und `neumann` folgendermaßen zu: Die Menge der Knoten $\mathcal{N} = \{v_1, \dots, v_N\}$ speichern wir als $N \times 3$ Array `coordinates`. Dabei entspricht die ℓ-te Zeile von `coordinates` genau dem Knoten $v_\ell = (x_\ell, y_\ell, z_\ell) \in \mathbb{R}^3$:

$$\texttt{coordinates(}\ell\texttt{,:)= [}x_\ell\ y_\ell\ z_\ell\texttt{]}. \tag{3.1.1}$$

Die Reihenfolge der Knoten ist dabei vorerst gleichgültig. Sobald wir das Array der Knoten generiert haben, können wir die Menge der Tetraeder $\mathcal{T} = \{T_1, \dots, T_M\}$ durch ein $M \times 4$ Array `elements` darstellen. Jede Zeile von `elements` entspricht einem Tetraeder T, indem

sie die vier Eckpunkte in Form von Indizes des Arrays `coordinates` kodiert. Der Tetraeder $T_\ell = \operatorname{conv}\{v_i, v_j, v_k, v_m\} \in \mathcal{T}$ wird also als

$$\texttt{elements(}\ell\texttt{,:) = [i,j,k,m]} \tag{3.1.2}$$

gespeichert. Damit ergibt sich beispielsweise, dass `coordinates(elements(`ℓ`,:),:)` eine 4×3 Matrix ist, deren Zeilen die Eckpunkte des Tetraeders T_ℓ darstellen. Die Reihenfolge der Knoten innerhalb der Zeile, sowie die Reihenfolge der Tetraeder ist vorerst noch beliebig. Für das Verfeinerungsschema, das wir etwas später behandeln, werden wir diese Bedingung allerdings noch etwas verschärfen müssen (siehe Definition 5.3.9). Zuletzt betrachten wir noch Γ_D und Γ_N. Die Definition der regulären Triangulierung garantiert, dass jede Randfläche $F \in \mathcal{F}$ mit $F \subseteq \Gamma$ entweder ganz in $\overline{\Gamma}_N$ oder in $\overline{\Gamma}_D$ enthalten ist. Da natürlich $\bigcup_{F\in\mathcal{F}} F \supseteq \Gamma$ gilt, speichern wir die Ränder Γ_D und Γ_N über die zugehörigen Randflächen. Dies geschieht auch im Simplex-Vertex-Format. Für eine Fläche $F_\ell = \operatorname{conv}\{v_i, v_j, v_k\} \in \mathcal{F}$ mit $F_\ell \subseteq \overline{\Gamma}_D$ speichern wir also

$$\texttt{dirichlet(}\ell\texttt{,:) = [i,j,k]}$$

Analog generieren wir `neumann` aus Γ_N. Auch bei diesen Randflächen ist uns wie bei den Tetraedern die Reihenfolge der Knoten vorerst egal.

3.2 Erste Implementierung des Laplace-Solvers

Bevor wir uns genauer mit der Implementierung beschäftigen, müssen wir uns zuerst überlegen, wie wir die Integrale aus den Formeln (1.3.6) und (1.3.7) berechnen. Für die rechte Seite **b** haben wir alles Nötige zur Verfügung, um eine Gauß-Quadratur durchzuführen. Für die Einträge der Steifigkeitsmatrix fehlt uns aber noch die genaue Kenntnis der Gradienten der Hutfunktionen. Dieses Wissen liefert uns folgendes Lemma, das auch ähnlich in [1, Abschnitt 5] zu finden ist.

Lemma 3.2.1. *Sei $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ ein nicht-entarteter Tetraeder und weiters V_1, V_2, V_3, V_4 die zu den Eckpunkten gehörigen Hutfunktionen auf T. Dann lässt sich die Matrix $A \in \mathbb{R}^{4\times4}$, definiert durch*

$$A_{jk} = \int_T \nabla V_j \cdot \nabla V_k \, dx, \tag{3.2.1}$$

darstellen als

$$A = \frac{1}{6} |\det(B)| G G^T. \tag{3.2.2}$$

Die Matrizen $B \in \mathbb{R}^{4\times4}$ und $G \in \mathbb{R}^{4\times4}$ sind hierbei durch

$$B := \begin{pmatrix} 1 & 1 & 1 & 1 \\ v_1 & v_2 & v_3 & v_4 \end{pmatrix} \text{ und } G := B^{-1} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.2.3}$$

definiert.

Beweis. Zuerst stellen wir fest, dass die Matrix B für jeden nicht-entarteten Tetraeder T invertierbar ist. Denn für das Volumen gilt $|T| = |\det(B)|/6$ wegen der Gleichheit

$$|\det(B)| = \left|\det \begin{pmatrix} 1 & 0 & 0 & 0 \\ v_1 & v_2 - v_1 & v_3 - v_1 & v_4 - v_1 \end{pmatrix}\right| = |\det(v_2 - v_1, v_3 - v_1, v_4 - v_1)| = 3!|T|, \tag{3.2.4}$$

da in der letzten Gleichheit die Determinante genau das Volumen des von $\{v_2-v_1, v_3-v_1, v_4-v_1\}$ aufgespannten Parallelepipeds liefert. Wir führen nun die Baryzentrischen Koordinaten $\lambda(x) \in \mathbb{R}^4$ als Lösung des Gleichungssystem $B\lambda(x) = \begin{pmatrix} 1 \\ x \end{pmatrix}$ ein. Diese Bedingung ist äquivalent zu $\sum_{j=1}^4 \lambda_j(x) = 1$ und $x = \sum_{j=1}^4 \lambda_j(x) v_j$, was insbesondere $\lambda_j(v_k) = \delta_{jk}$ bedeutet. Wenn wir nun $\lambda(x)$ in folgender Form anschreiben

$$\lambda(x) = \begin{pmatrix} \lambda_1(x) \\ \lambda_2(x) \\ \lambda_3(x) \\ \lambda_4(x) \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \text{ wobei } C := B^{-1} \text{ ist,} \tag{3.2.5}$$

erkennen wir, dass die λ_j affine Abbildungen darstellen. Zusammen mit der Eigenschaft, dass $\lambda_j(v_k) = \delta_{jk}$, gilt erhalten wir also die Gleichheit von $\lambda_j = V_j$ auf T. Zu den affinen Abbildungen lassen sich nun leicht die Gradienten bestimmen.

$$\begin{pmatrix} \nabla\lambda_1(x) \\ \nabla\lambda_2(x) \\ \nabla\lambda_3(x) \\ \nabla\lambda_4(x) \end{pmatrix} = \begin{pmatrix} c_{12} & c_{13} & c_{14} \\ c_{22} & c_{23} & c_{24} \\ c_{32} & c_{33} & c_{34} \\ c_{42} & c_{43} & c_{44} \end{pmatrix} = C \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = G \tag{3.2.6}$$

Da nun also die Zeilen von G genau die Gradienten der Hutfunktionen sind, erhalten wir

$$\nabla V_j \cdot \nabla V_k = (GG^T)_{jk}, \tag{3.2.7}$$

und, weil $\nabla V_j \cdot \nabla V_k$ konstant auf T ist, schließlich

$$A_{jk} = \int_T \nabla V_j \cdot \nabla V_k \, dx = |T| \nabla V_j \cdot \nabla V_k = \frac{1}{6} |\det(B)| (GG^T)_{jk}. \tag{3.2.8}$$

Damit ist unser Lemma gezeigt. □

Wir werden nun eine erste Implementierung des Verfahrens besprechen. (Siehe dazu Listing 3) Für den Aufbau der Steifigkeitsmatrix werden wir das vorangegangene Lemma benützen, die rechte Seite bestimmen wir mittels Gauß-Quadratur.

Listing 3: LAPLACE-SOLVER

```
function [x,energy] = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD)
quaddeg = 3;
nC = size(coordinates,1);
nE = size(elements,1);
x = zeros(nC,1);
%*** Steifigkeitsmatrix anlegen
A = sparse(nC,nC);
for i = 1:nE
    nodes = elements(i,:);
    B = [1 1 1 1; coordinates(nodes,:)'];
    grad = B \ [0 0 0; 1 0 0; 0 1 0; 0 0 1];
    A(nodes,nodes) = A(nodes,nodes) + abs(det(B))*grad*grad'/6;
end
%*** Den Dirichlet-Knoten die Werte zuweisen
dirichlet = unique(dirichlet);
x(dirichlet) = feval(uD,coordinates(dirichlet,:));
%*** Bestimmung der rechten Seite des linearen Gleichungssystems
```

```
18 b = -A*x;
19 for i = 1:nE
20     nodes = elements(i,:);
21     %*** Bestimmung der Integrale
22     [XYZ,W,~,V] =  tetquad(quaddeg,coordinates(nodes,:));
23     integrals = W*(repmat(feval(f,XYZ),1,4).*V);
24     b(nodes) = b(nodes) + integrals';
25 end
26 for i = 1:size(neumann,1)
27     nodes = neumann(i,:);
28     %*** Bestimmung der Integrale
29     [XYZ,W,~,V] = triquad(quaddeg,coordinates(nodes,:));
30     integrals = W*(repmat(feval(g,XYZ),1,3).*V);
31     b(nodes) = b(nodes) + integrals';
32 end
33 %*** Berechnung der diskreten Loesung, sowie deren Energie
34 freenodes = setdiff(1:nC, dirichlet);
35 x(freenodes) = A(freenodes,freenodes)\b(freenodes);
36 energy = x'*A*x;
```

- Zeile 1: Die Funktion `solveLaplace` übernimmt eine reguläre Triangulierung $\mathcal{T}$ sowie die Funktion f, die Neumann-Daten g und die Dirichlet-Daten u_D. Diese Funktionen können dabei jeweils entweder als Function-Handle oder als String des entsprechenden Funktionsnamen übergeben werden. Entscheidend ist, dass `f`, `g` und `uD` so realisiert sind, dass gleichzeitig n Punkte in Form einer Matrix $\xi \in \mathbb{R}^{n\times 3}$ übergeben und als Spaltenvektor $y \in \mathbb{R}^{n\times 1}$ ausgewertet werden können. Das Verfahren liefert schließlich den Koeffizientenvektor $\mathbf{x}$ der diskreten Lösung $U \in \mathcal{S}^1(\mathcal{T})$ sowie die Energie $\|\nabla U\|^2_{L^2(\Omega)} = \sum_{j,k=0}^{N} \mathbf{x}_j \mathbf{x}_k \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \mathbf{x}^T \mathbf{A} \mathbf{x}$
- Zeilen 7–13: Hier wird die Steifigkeitsmatrix $\mathbf{A} \in \mathbb{R}^{N\times N}_{\text{sym}}$, wie in (1.3.7) beschrieben, aufgebaut. Dabei wird ausgenutzt, dass ein Summand des Eintrags A_{jk} gegeben durch
$$\int_T \nabla V_j \cdot \nabla V_k \, dx = |T| \, \nabla V_j|_T \cdot \nabla V_k|_T \tag{3.2.9}$$
verschwindet, falls T einen der Knoten v_j oder v_k nicht enhält. Es müssen also nur Integrale bestimmt werden, deren entsprechende Knoten v_j und v_k zu T gehören. Somit kann die Matrix dadurch aufgebaut werden, dass für jedes Element $T = \operatorname{conv}\{v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}\} \in \mathcal{T}$, die zu den Indizes i_1, i_2, i_3, i_4 gehörige 4×4 Untermatrix geändert wird. Diese Untermatrizen werden durch Anwendung von Lemma 3.2.1 aufgebaut.
- Zeilen 15–16: Die Einträge von $x \in \mathbb{R}^N$, die den Dirichlet-Knoten entsprechen werden wie in (1.3.5) beschrieben durch `uD` bestimmt.
- Zeilen 18–35: Hier wird die rechte Seite $\mathbf{b} \in \mathbb{R}^N$ aus Gleichung (1.3.6) aufgebaut. Wir beginnen damit den Beitrag der Dirichlet-Daten hinzuzufügen. In den Zeilen 19–25 addieren wir elementweise den Beitrag von
$$\int_\Omega f V_j \, dx = \sum_{T\in\mathcal{T}} \int_T f V_j \, dx. \tag{3.2.10}$$
Dabei sind alle Integrale Null, deren Tetraeder T den Knoten v_j nicht enthalten. Damit ergibt sich für jedes Element lediglich eine Änderung von vier Komponenten von $\mathbf{b}$. Wir berechnen die Integrale mit einer mehrdimensionalen Gauß-Quadratur. Die Funktion

`tetquad` liefert dabei neben den Gauß-Knoten `XYZ` und Gauß-Gewichten `W` den Array der baryzentrischen Koordinaten `V` zurück. Ihre Funktionsweise wird im Abschnitt 2 genauer behandelt. Jede Zeile von baryzentrischen Koordinaten `V(ℓ,:)` entspricht genau den kartesischen Koordinaten `XYZ(ℓ,:)`. Wie wir im Beweis von Lemma 3.2.1 gesehen haben, entsprechen diese genau den Hutfunktionen auf T. Damit berechnen wir

$$\int_T f V_j dx \approx \sum_{k=1}^{q^3} w_k f(x_k) V_j(x_k) \tag{3.2.11}$$

mit q der Anzahl Interpolationsknoten je Dimension und den transformierten Gauß-Knoten x_k und Gauß-Gewichten w_k. In den Zeilen 26–32 werden schließlich die Beiträge des Neumannrandes hinzugefügt.

$$\int_{\Gamma_N} g V_j \, ds = \sum_{F \in \mathcal{F}_N} \int_F g V_j \, ds, \text{ wobei } \mathcal{F}_N := \{F \in \mathcal{F} : F \subseteq \overline{\Gamma}_N\} \tag{3.2.12}$$

Auch diese werden ähnlich wie zuvor mittels Gauß-Quadratur berechnet. Siehe dazu Listing 2.

- Zeilen 34–36: Nun muss nur mehr das Gleichungssystem gelöst werden. Dazu werden zuerst die Indizes der freien Knoten $v_j \notin \overline{\Gamma}_D$ bestimmt (Zeile 34) und durch Lösung des entsprechenden Gleichungssystems die Werte $\mathbf{x}_j$ mit $v_j \notin \overline{\Gamma}_D$ berechnet (Zeile 35). Wir erhalten den Koeffizientenvektor $\mathbf{x}$ der Approximation $U \in S^1(\mathcal{T})$ aus Gleichung (1.3.5).

Die Erwartungen, die man auf den ersten Blick an die Laufzeit der Funktion `solveLaplace` (exklusive der Lösung des linearen Gleichungssystems) stellen möge, nämlich Linearität in der Anzahl der Tetraeder $\#\mathcal{T}$, werden bei dieser Implementierung nicht erfüllt. Vielmehr kann eine quadratische Laufzeit beobachtet werden. Siehe dazu Abbildung 7 auf Seite 30.

3.3 Gründe für die Ineffizienz & Verbesserungsvorschlag

In diesem Abschnitt werden wir besprechen, warum sich die Laufzeit der Funktion `solveLaplace` aus Listing 3 quadratisch in $M = \#\mathcal{T}$ verhält, und eine verbesserte Version vorschlagen. Eine Laufzeitanalyse in MATLAB zeigt, dass der Aufbau der Matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ am zeitaufwändigsten ist. Das liegt an der Art und Weise, wie MATLAB schwachbesetzte `sparse`-Matrizen speichert. Dies geschieht im sogenannten *CCS-Format* (compressed column storage), das auch unter *Harwell-Boeing-Format* bekannt ist. Mit $n := |\{(i,j) : \mathbf{A}_{ij} \neq 0\}|$ der Anzahl der Nichtnulleinträge der Matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ werden dafür die Vektoren $a \in \mathbb{R}^n$, $I \in \mathbb{N}^n$, $J \in \mathbb{N}^{N+1}$ folgendermaßen gespeichert:

- a enthält spaltenweise alle Nichtnulleinträge der Matrix $\mathbf{A}$,
- I enthält die zu den Einträgen aus a gehörigen Zeilenindizes, und
- J gibt an wo die einzelnen Spalten von $\mathbf{A}$ im Vektor a beginnen. Für $1 \leq k \leq N$ ist J_k der Index (im Vektor a) des ersten Nichtnulleintrags der k-ten Spalte von $\mathbf{A}$. Für $k = N+1$ gilt $J_{N+1} := N+1$.

Im Gegensatz zu vollbesetzten Matrizen ist hier das Einfügen von neuen Nichtnulleinträgen nicht in $\mathcal{O}(1)$ durchführbar, es müssen nämlich, neben dem Anlegen von zusätzlichem Speicher für die neuen Einträge, die Speicherungsvektoren neu sortiert werden. Mit i der Anzahl der aktuellen Nichtnulleinträge liegt der Aufwand für das Einfügen eines Eintrags somit bei etwa $\mathcal{O}(i \log i)$,

was bei Einfügen von n Einträgen zu einer Laufzeit von mehr als $\mathcal{O}(n^2)$ führt. MATLAB bietet allerdings die Möglichkeit, den Aufbau einer schwachbesetzten Matrix effizient durchzuführen. Dazu wird die Matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ mit n Nichtnulleinträgen zuerst im erweiterten *Koordinatenformat* aufgebaut. Das Koordinatenformat solch einer Matrix sind die drei Vektoren $a \in \mathbb{R}^n$, $I \in \mathbb{N}^n$, $J \in \mathbb{N}^n$, die so zu lesen sind:

- a enthält die Nichtnulleinträge der Matrix $\mathbf{A}$ in beliebiger Reihenfolge,
- I enthält die zu den Einträgen aus a gehörigen Zeilenindizes, und
- J enthält die zu den Einträgen aus a gehörigen Spaltenindizes.

Dieses Format kann nun erweitert werden, indem wir das mehrfache Auftreten von Indexpaaren $(i,j) = (I_{k_1}, J_{k_1}) = (I_{k_2}, J_{k_2})$ zulassen. Wir interpretieren dies dann dadurch, dass die entsprechenden Einträge aufsummiert werden. Wenn nun diese drei Vektoren `a`, `I` und `J` vorliegen, kann die zugehörige schwachbesetzte Matrix mit dem Befehl

```
A = sparse(I,J,a,N,N)
```

aufgebaut werden. Dieser Befehl ist äquivalent zum folgenden Code, er ist allerdings intern effizient implementiert.

```
A = sparse(N,N);
for k = 1:length(a)
    A(I(k),J(k)) = A(I(k),J(k))+a(k);
end
```

Mit obigem `sparse`-Befehl können wir also die Einträge unserer Steifigkeitsmatrix ohne Bedenken tetraederweise im erweiterten Koordinatenformat berechnen, da die dabei entstehenden mehrfachen Indexpaare durch den `sparse`-Befehl wie gewünscht aufaddiert werden. Nun hat aber die Erstellung der Matrix einen deutlich geringeren Aufwand. Zur effizienten Umwandlung in das CCS-Format ist nämlich lediglich ein Sortieren der Einträge notwendig, sowie die Umwandlung des Vektors J, was aber mit linearem Zeitaufwand bewerkstelligt werden kann. Der für die Sortierung benötigte Aufwand dominiert also und liefert Komplexität $\mathcal{O}(n \log n)$. Dies wird auch durch die Laufzeit der Listing 4 bestätigt, in der wir zeigen, wie die Matrix in dieser Form aufgebaut wird.

Listing 4: STEIFIGKEITSMATRIX FASTLINEAR

```
%*** Steifigkeitsmatrix anlegen
nE = size(elements,1);
I = zeros(16*nE,1);
J = zeros(16*nE,1);
A = zeros(16*nE,1);
for i = 1:nE
    nodes = elements(i,1:4);
    B = [1 1 1 1; coordinates(nodes,:)'];
    grad = B \ [0 0 0; 1 0 0; 0 1 0; 0 0 1];
    idx = (16*(i-1)+1):(16*i);
    tmp = [1;1;1;1]*nodes;
    I(idx) = reshape(tmp',16,1);
    J(idx) = reshape(tmp, 16,1);
    A(idx) = abs(det(B))/6*reshape(grad*grad',16,1);
end
A = sparse(I,J,A,nC,nC);
```

- Zeilen 2–5: Die Vektoren I, J und a werden angelegt. Wie in Listing 3 erkennbar, werden zu jedem Element $T \in \mathcal{T}$ genau 16 Einträge der Matrix $\mathbf{A}$ aktualisiert. Die Vektoren I, J und a haben also die Länge $16M$ mit $M = \#\mathcal{T}$ der Anzahl der Elemente.
- Zeilen 10–14: Die Einträge werden in Blöcken zu 16 Einträgen geändert. Bemerke dazu, dass die 4×4 Matrizen C :=`tmp`, D :=`tmp'` und E :=`abs(det(B))/6*(grad*grad')` erfüllen, dass die Tupel (C_{jk}, D_{jk}, E_{jk}) für $j, k = 1 \dots 4$ genau die 16 zu T gehörenden Einträge im Koordinaten-Format darstellen.
- Zeile 16: Die schwachbesetzte Matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ wird mit den drei berechneten Vektoren bestimmt.

Wenn wir nun die Laufzeiten der beiden Implementierungen vergleichen, beobachten wir, dass sich der Aufbau der Matrix mit der verbesserten Variante fastlinear in $M = \#\mathcal{T}$ verhält, die ursprüngliche Variante nur quadratisch. Siehe dazu Abbildung 7 auf Seite 30.

4 Eine verbesserte MATLAB-Implementierung

In diesem Abschnitt werden wir die Laufzeit der Funktion `solveLaplace` weiter verbessern. Da häufige Funktionsaufrufe in MATLAB den Code stark verlangsamen, werden wir die `for`-Schleifen durch Vektorarithmetik ersetzen. Dazu listen wir hier einige Befehle auf, die entscheidend für die Implementierung sind.

4.1 Wichtige MATLAB-Befehle

- Obwohl für die Berechnung eines Skalarproduktes keinerlei außergewöhnliche Funktionen benötigt werden, verwenden wir ob der besseren Lesbarkeit den Befehl `dot`. Mittels `S = dot(A,B,2)` kann zu zwei Matrizen $A, B \in \mathbb{R}^{m \times n}$ der Vektor $S \in \mathbb{R}^m$ der Skalarprodukte der Zeilen berechnet werden.
- Zur Berechnung des Kreuzproduktes verwenden wir die Funktion `cross`. Zu zwei $n \times 3$ Matrizen A und B liefert der Befehl `C = cross(A,B,2)` die $n \times 3$ Matrix C, deren Zeilen genau die Kreuzprodukte entsprechender Zeilen von A und B sind.
- Häufig wird auch der Befehl `B = reshape(A,m,n)` verwendet, der zu einer beliebigen Matrix mit mn Einträgen die Matrix $B \in \mathbb{R}^{m \times n}$ liefert, deren spaltenweise Speicherung mit der von A übereinstimmt, d.h. `A(:)` ist genau `B(:)`. Weiters kann die Funktion auch durch `B = reshape(A,[],n)` oder `B = reshape(A,m,[])` aufgerufen werden, wobei MATLAB die fehlende Dimension selbst berechnet.
- Um die Gauß-Knoten für jeden Tetraeder der Triangulierung zu generieren, gehen wir nun so vor: Wir generieren lediglich die baryzentrischen Koordinaten der gewünschten Quadraturpunkte und wandeln diese dann mit dem Befehl `baryToCart` in kartesische Koordinaten um. Dies geschieht seit Version R2009a mit dem Befehl `XC = baryToCart(TR, SI, B)`. Dabei ist `TR` als ein Objekt der Klasse `TriRep`, eine Struktur die unsere Arrays `elements` und `coordinates` zusammenfasst. Mit k der Anzahl der zu konvertierenden Punkte ist `SI` ein $k \times 1$ Array und `B` ein $k \times 4$ Array. Dabei gibt jeder Eintrag `SI(i)` den Index des Tetraeders an, sodass `XC(i,1:3)` die kartesischen Koordinaten des zum Tetraeder `elements(SI(i),:)` gehörigen Punktes mit baryzentrischen Koordinaten `B(i,1:4)` sind.
- Für eine Matrix $A \in \mathbb{R}^{m \times n}$ erzeugt die Funktion `B = repmat(A,M,N)` eine $M \times N$ Blockmatrix $B \in \mathbb{R}^{mM \times nN}$, deren Blöcke Kopien der Matrix A sind.

- Schließlich verwenden wir noch den Befehl `b = accumarray(subs,val,[m n])`. Dieser akkumuliert die Werte aus `val` an die Stellen von `subs` in ein Array `b = zeros(m,n)`. In eine `for`-Schleife übersetzt würde der Befehl `b = accumarray(subs,val,[m 1])` für Spaltenvektoren `subs` und `val` etwa so aussehen:

```
b = zeros(m,1);
for i = 1:length(subs)
    b(subs(i)) = b(subs(i)) + val(i);
end
```

4.2 Berechnung der Steifigkeitsmatrix

Zur Berechnung der Gradienten werden wir nun folgendes Lemma verwenden:

Lemma 4.2.1. *Sei $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ ein nicht-entarteter Tetraeder und n_i normal auf der Seitenfläche $\operatorname{conv}\{v_j : j \neq i\}$, dann lassen sich die Gradienten der Hutfunktionen durch*

$$\nabla V_i|_T = \frac{1}{\langle n_i, v_i - v_j\rangle} n_i \text{ mit } j \neq i \tag{4.2.1}$$

berechnen.

Beweis. Sei zunächst A_i die von den Punkten $\mathcal{N}_T \setminus \{v_i\}$ aufgespannte affine Hyperebene und H_i die Hyperebene der Richtungsvektoren von A_i. Die affine Ebene können wir mit $a \in A_i$ durch $A_i = a + H_i$ darstellen. Mit einem beliebigen Vektor n_i, der normal auf H_i steht, definieren wir die Abbildung:

$$\tilde{V}_i(x) := \frac{\langle n_i, x - a\rangle}{\langle n_i, v_i - a\rangle} = \frac{\langle n_i, x\rangle}{\langle n_i, v_i - a\rangle} - \frac{\langle n_i, a\rangle}{\langle n_i, v_i - a\rangle}. \tag{4.2.2}$$

Diese Abbildung ist affin und erfüllt $\tilde{V}_i(v_j) = 0$ für $j \neq i$ und $\tilde{V}_i(v_i) = 1$. Auf T eingeschränkt ist sie also gleich unserer Hutfunktion $V_i|_T = \tilde{V}_i|_T$. Mit dieser expliziten Darstellung lässt sich der Gradient nun leicht berechnen:

$$\nabla\tilde{V}_i(x) = \frac{1}{\langle n_i, v_i - a\rangle}\nabla(\langle n_i, x\rangle - \langle n_i, a\rangle) = \frac{1}{\langle n_i, v_i - a\rangle}\nabla\langle n_i, x\rangle = \frac{1}{\langle n_i, v_i - a\rangle} n_i. \tag{4.2.3}$$

Hierbei kann $a \in A_i$ als einer der übrigen drei Punkte $\{v_j : j \neq i\} \subseteq A_i$ gewählt werden. Damit ist das Lemma gezeigt. □

Dieses Lemma können wir nun verwenden um damit die Einträge der Steifigkeitsmatrix zu berechnen.

Lemma 4.2.2. *Sei $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ ein nicht-entarteter Tetraeder. Mit der Vereinbarung, dass*

$$j \oplus k := ((j + k - 1) \bmod 4) + 1, \tag{4.2.4}$$

also beispielsweise $4 \oplus 1 = 1 = 3 \oplus 2$ gilt, seien die Normalvektoren n_i definiert durch:

$$n_i := (-1)^i (v_{i\oplus 2} - v_{i\oplus 1}) \times (v_{i\oplus 3} - v_{i\oplus 1}) \quad \text{für } i \in \{1, 2, 3, 4\}. \tag{4.2.5}$$

Die Einträge der lokalen Steifigkeitsmatrix lassen sich dann durch

$$A_{jk} = \frac{\langle n_j, n_k\rangle}{36|T|} \tag{4.2.6}$$

berechnen.

Beweis. Mit Lemma 4.2.1 lassen sich die Einträge der lokalen Steifigkeitsmatrix folgenderweise berechnen:

$$A_{jk} = \int_T \langle \nabla V_j, \nabla V_k \rangle \, dx = \frac{|T| \langle n_j, n_k \rangle}{\langle n_j, v_j - v_{j\oplus 1} \rangle \langle n_k, v_k - v_{k\oplus 1} \rangle}. \tag{4.2.7}$$

Nun ist Folgendes entscheidend: Wählt man als n_i die obigen Kreuzprodukte, dann gilt für alle $i, j \in \{1, 2, 3, 4\}$:

$$|\langle n_i, v_i - v_{i\oplus 1} \rangle| = 6|T| \quad \text{und} \tag{4.2.8}$$

$$\operatorname{sgn} \langle n_i, v_i - v_{i\oplus 1} \rangle = \operatorname{sgn} \langle n_j, v_j - v_{j\oplus 1} \rangle. \tag{4.2.9}$$

Es sind also die vier Normalvektoren entweder allesamt nach außen oder alle nach innen orientiert und das Volumen lässt sich auf genannte Art durch den Normalvektor berechnen. Wir definieren P_i^2 als das von $(v_{i\oplus 2} - v_{i\oplus 1})$ und $(v_{i\oplus 3} - v_{i\oplus 1})$ aufgespannten Parallelogramm und P_i^3 als das von $(v_{i\oplus 2} - v_{i\oplus 1})$, $(v_{i\oplus 3} - v_{i\oplus 1})$ und $(v_{i\oplus 4} - v_{i\oplus 1})$ aufgespannte Parallelepiped. Nun entspricht die Länge des Vektors n_i genau dem Flächeninhalt von P_i^2. Weiters ist $\left\langle \frac{1}{|n_i|} n_i, v_i - v_{i\oplus 1} \right\rangle$ die Höhe von P_i^3, wenn man P_i^2 als Grundfläche betrachtet. Damit gilt

$$|\langle n_i, v_i - v_{i\oplus 1} \rangle| = |n_i| \left| \left\langle \frac{1}{|n_i|} n_i, v_i - v_{i\oplus 1} \right\rangle \right| = |P_i^3| = 6|T|. \tag{4.2.10}$$

Zur Orientierung der Normalvektoren sei angemerkt, dass es für einen regelmäßigen Tetraeder nur zwei verschiedene Nummerierungen der Knoten mit den Ziffern $\{1, 2, 3, 4\}$ gibt, die nicht durch Drehungen ineinander überführbar sind. Man könnte von einem „linksdrehenden“ und einem „rechtsdrehenden“ Tetraeder sprechen.

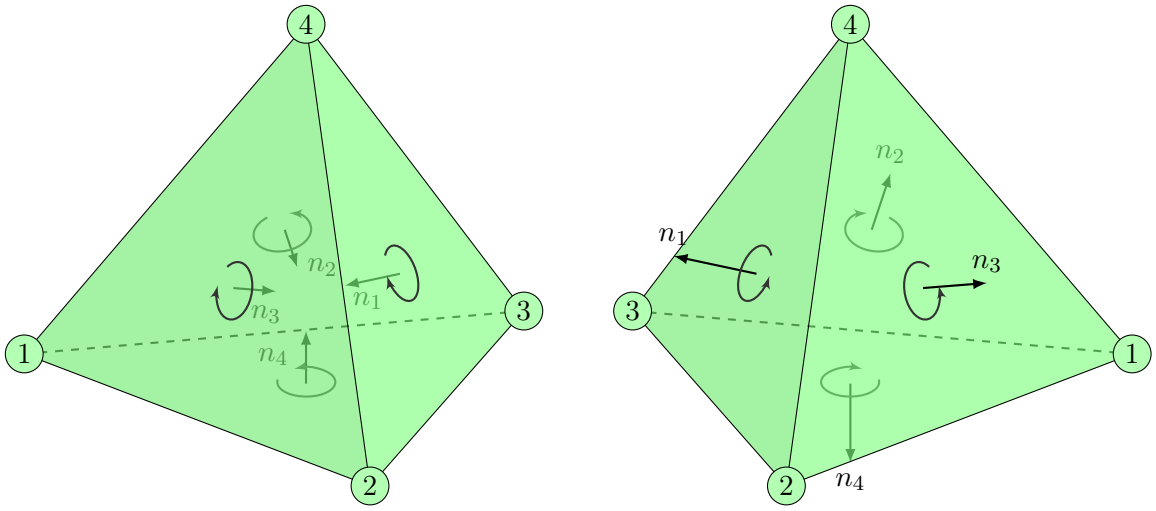


Abbildung 6: Die möglichen Orientierungen der Normalvektoren aus Lemma 4.2.2. Entweder sind alle nach außen, oder alle nach innen orientiert. Jede Markierung der Knoten mit den Zahlen 1–4 führt auf einen dieser zwei Fälle.

Für diese beiden ausgezeichneten Tetraeder lässt sich die Orientierungseigenschaft (4.2.9) dann mittels Einsetzen in die Definition der Normalvektoren überprüfen. Dazu sei auf Abbildung 6 verwiesen. Diese Eigenschaft wenden wir nun auf Gleichung (4.2.7) an und erhalten:

$$A_{jk} = \frac{|T| \langle n_j, n_k \rangle}{\langle n_j, v_j - v_{j\oplus 1} \rangle \langle n_k, v_k - v_{k\oplus 1} \rangle} = \frac{|T| \langle n_j, n_k \rangle}{\langle n_1, v_1 - v_2 \rangle^2} = \frac{\langle n_j, n_k \rangle}{36|T|}. \tag{4.2.11}$$

Dies schließt unseren Beweis. □

4.3 Vektorisierte Implementierung

In diesem Abschnitt wird nun die vektorisierte Version des Laplace-Solvers aus Listing 5 besprochen.

Listing 5: Laplace-Solver vektorisiert

```
function [x,energy] = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD)
quaddeg = 3;
nC = size(coordinates,1);
nE = size(elements,1);
nN = size(neumann,1);
x = zeros(nC,1);
%*** Berechnung der Normalvektoren
X1 = coordinates(elements(:,1),:);
X2 = coordinates(elements(:,2),:);
X3 = coordinates(elements(:,3),:);
X4 = coordinates(elements(:,4),:);
N1 = -cross(X3-X2,X4-X2,2);
N2 =  cross(X4-X3,X1-X3,2);
N3 = -cross(X1-X4,X2-X4,2);
N4 =  cross(X2-X1,X3-X1,2);
%*** Bestimmung der Volumina der Tetraeder
volumes = (1/6)*abs(dot(N1,X1-X2,2));
%*** Berechnung der Steifigkeitsmatrix
A = repmat(1./(36*volumes),16,1)...
     .*[dot(N1,N1,2);dot(N1,N2,2);dot(N1,N3,2);dot(N1,N4,2);...
        dot(N2,N1,2);dot(N2,N2,2);dot(N2,N3,2);dot(N2,N4,2);...
        dot(N3,N1,2);dot(N3,N2,2);dot(N3,N3,2);dot(N3,N4,2);...
        dot(N4,N1,2);dot(N4,N2,2);dot(N4,N3,2);dot(N4,N4,2)];
idx = ones(4,1)*(1:4);
I = reshape(elements(:,idx), 16*nE,1);
J = reshape(elements(:,idx'),16*nE,1);
A = sparse(I,J,A,nC,nC);
%*** Den Dirichlet-Knoten die Werte zuweisen
dirichlet = unique(dirichlet);
x(dirichlet) = feval(uD,coordinates(dirichlet,:));
%*** Bestimmung der rechten Seite des linearen Gleichungssystems
  %* Integration von f V_j ueber Omega
TetMesh = TriRep(elements,coordinates);
[~,W,~,V] = tetquad(quaddeg,[eye(3);zeros(1,3)]);
Vrep = repmat(V,nE,1);
XYZ = baryToCart(TetMesh, reshape(repmat(1:nE,quaddeg.^3,1),[],1), Vrep);
integrals = repmat(volumes,1,4)...
          .*reshape(6*W*(reshape(repmat(feval(f,XYZ),1,4).*Vrep,[],4*nE)),nE,[]);
b = accumarray(elements(:),integrals(:),[nC 1]) - A*x;
  %* Integration von g V_j ueber Gamma_N
if (nN~=0)
  N = size(neumann,1);
  TriMesh = TriRep(neumann,coordinates);
  S21 = coordinates(neumann(:,2),:) - coordinates(neumann(:,1),:);
  S31 = coordinates(neumann(:,3),:) - coordinates(neumann(:,1),:);
  areas = 1/2*sqrt(sum(cross(S21,S31,2).^2,2));
  [~,W,~,V] = triquad(quaddeg,diag([1,1,0]));
  Vrep = repmat(V,nN,1);
  XYZ = baryToCart(TriMesh, reshape(repmat(1:nN,quaddeg.^2,1),[],1), Vrep);
  integrals = repmat(areas,1,3)...
          .*reshape(2*W*(reshape(repmat(feval(g,XYZ),1,3).*Vrep,[],3*nN)),nN,[]);
  b = b + accumarray(neumann(:),integrals(:),[nC 1]);
end
```

```
54 %*** Berechnung der diskreten Loesung, sowie deren Energie
55 freenodes = setdiff(1:nC, dirichlet);
56 x(freenodes) = A(freenodes,freenodes)\b(freenodes);
57 energy = x'*A*x;
```

- Zeile 1: Die Funktion `solveLaplace` übernimmt eine reguläre Triangulierung $\mathcal{T}$ sowie die Funktion f, die Neumann-Daten g und die Dirichlet-Daten u_D. Diese Funktionen können dabei jeweils entweder als Function-Handle oder als String des entsprechenden Funktionsnamen übergeben werden. Entscheidend ist, dass `f`, `g` und `uD` so realisiert sind, dass gleichzeitig n Punkte in Form einer Matrix $\xi \in \mathbb{R}^{n\times 3}$ übergeben und als Spaltenvektor $y \in \mathbb{R}^{n\times 1}$ ausgewertet werden können. Das Verfahren liefert schließlich den Koeffizientenvektor $\mathbf{x}$ der diskreten Lösung $U \in \mathcal{S}^1(\mathcal{T})$ sowie die Energie $\|\nabla U\|^2_{L^2(\Omega)} = \sum_{j,k=0}^{N} \mathbf{x}_j \mathbf{x}_k \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \mathbf{x}^T \mathbf{A} \mathbf{x}$
- Zeilen 7–14: Hier werden die Normalvektoren der einzelnen Flächen wie in Lemma 4.2.2 berechnet. Die vier Vektoren sind dabei für jeden einzelnen Tetraeder entweder allesamt nach innen oder alle nach außen orientiert.
- Zeile 17: Speicherung der Volumina, die zur Generierung der Steifigkeitsmatrix, sowie zur Skalierung der Integrale in den Zeilen 36–37 benötigt werden im Array `volumes`.
- Zeilen 19–27: Hier werden die Einträge der Steifigkeitsmatrix berechnet. Dabei wird wieder das Lemma 4.2.2 verwendet.
- Zeilen 33–39: Die Beiträge der Funktion f werden mittels vektorisierter Gauß-Quadratur ausgewertet. Die Quadraturpunkte werden dazu in baryzentrischen Koordinaten generiert und anschließend für alle Tetraeder in kartesische Koordinaten umgewandelt. Die Quadratur geschieht anschließend vektorisiert. Dabei muss darauf geachtet werden, dass die Integrale richtig skaliert werden. Für alle Tetraeder werden dieselben Quadraturgewichte `W` mit `sum(W)` $= 1/6 = |T_{\text{ref}}|$ verwendet, deshalb müssen die Ergebnisse mit den Faktoren `6*volumes` skaliert werden. Der Vektor `b` wird mit erwähnten Integralen und den Dirichletdaten (Zeile 39) initialisiert.
- Zeilen 41–53: Für den Fall, dass der Neumann-Rand nichttrivial ist, werden hier die Beiträge der Funktion g berechnet. Dazu werden zuerst die Flächeninhalte der Dreiecke berechnet. Die Quadratur läuft ähnlich wie für die Funktion f ab. In diesem Fall summieren die Quadraturgewichte auf $1/2$ und müssen deshalb nachträglich mit `2*areas` multipliziert werden.

Ein Vergleich der Laufzeiten der hier dargestellten Varianten des Laplace-Solvers, sowie einer aktuellen Arbeit[12] von Rahman und Valdman ist nun Abbildung 7 zu entnehmen.

5 Verfeinern & Vergröbern

Es liegt in der Natur der Sache, ein gegebenes System nicht nur mit einer einzigen Diskretisierung des Gebietes zu lösen. Ziel ist es, Triangulierungen so zu wählen, dass der durch die Diskretisierung entstehende Fehler möglichst klein wird. Der naheliegende Weg, um das zu erreichen, ist das Verfeinern der ursprünglichen Triangulierung. Dabei sind neben uniformen Verfeinerungen, die alle Tetraeder teilen, besonders adaptive Verfahren von Interesse. Mit diesen kann die Diskretisierung dynamisch an den Stellen verbessert werden, an denen eine Verfeinerung am meisten nötig ist. Bei beiden Arten steht für die Analysis im Vordergrund, dass die Tetraeder

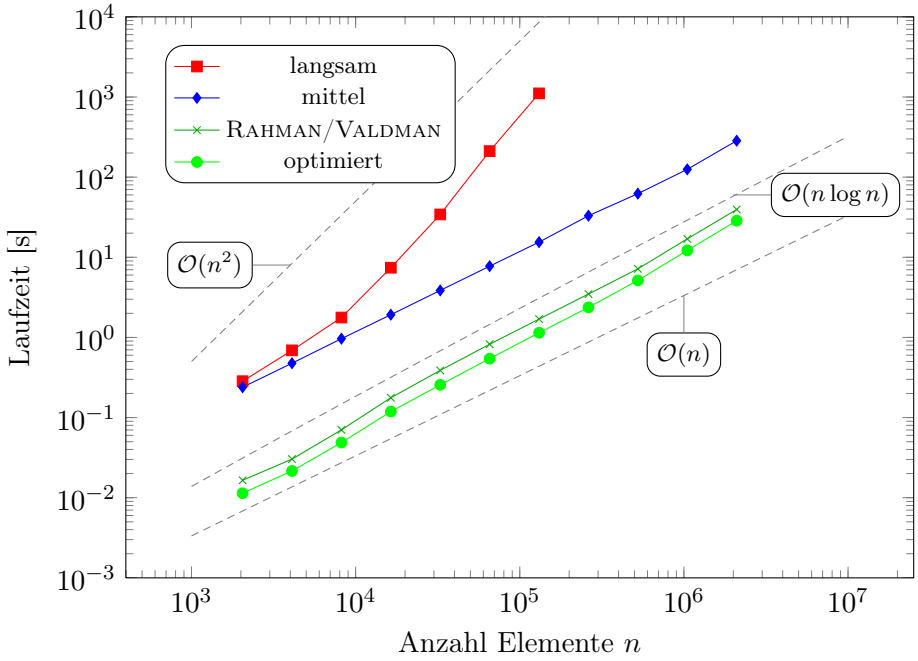


Abbildung 7: Laufzeiten zur Erstellung der Steifigkeitsmatrix der drei Implementierungen Listing 3(langsam), Listing 4(mittel) und Listing 5(optimiert) in Abhängigkeit der Anzahl der Elemente. Als Referenzwert ist auch die Implementierung von Rahman und Valdman aus [12] angegeben. Die verbesserten Varianten (mittel, optimiert, Rahman/Valdman) haben etwa fastlineare Laufzeit, die ursprüngliche Implementierung (langsam) quadratische.

nicht zu sehr entarten, man zum Beispiel die Anzahl der Ähnlichkeitsklassen nach oben abschätzen kann. Wir werden in diesem Abschnitt neben der Bestimmung von grundsätzlichen geometrischen Beziehungen und einem uniformen Verfeinerungsschema vor allem ein Verfahren zur adaptiven Verfeinerung besprechen.

5.1 Berechnung geometrischer Beziehungen

In Listing 6 klären wir die Generierung einer Hilfsstruktur, die zu einem gegebenen Tetraeder die benachbarten Tetraeder liefert. Für eine reguläre Triangulierung mit n Tetraedern liefert `provideNeighbors` eine $n \times 4$ Matrix mit den Indizes der benachbarten Tetraeder. Dabei gilt, dass der Tetraeder `element2neighbors(T,k)` an der Fläche von `T` liegt, die den Knoten `elements(T,k)` nicht enhält, also der dem Knoten `elements(T,k)` gegenüberliegenden Fläche. Diese Funktionalität ist seit der Version R2009a in Matlab als interne Routine `neighbors` verfügbar. Die Laufzeit der folgenden Implementierung verhält sich in Abhängigkeit der Anzahl der Elemente gleich wie die Matlab-Routine, ist jedoch etwa um den Faktor 4 langsamer (siehe Abbildung 8). Die Funktion `provideNeighbors` ist wie `neighbors` auch im zweidimensionalen Fall anwendbar und lässt sich leicht auf beliebige Dimensionen erweitern.

Listing 6: Geometriedaten für Nachbarschaftsrelation

```
1 function element2neighbors = provideNeighbors(elements)
2 nE = size(elements,1);
```

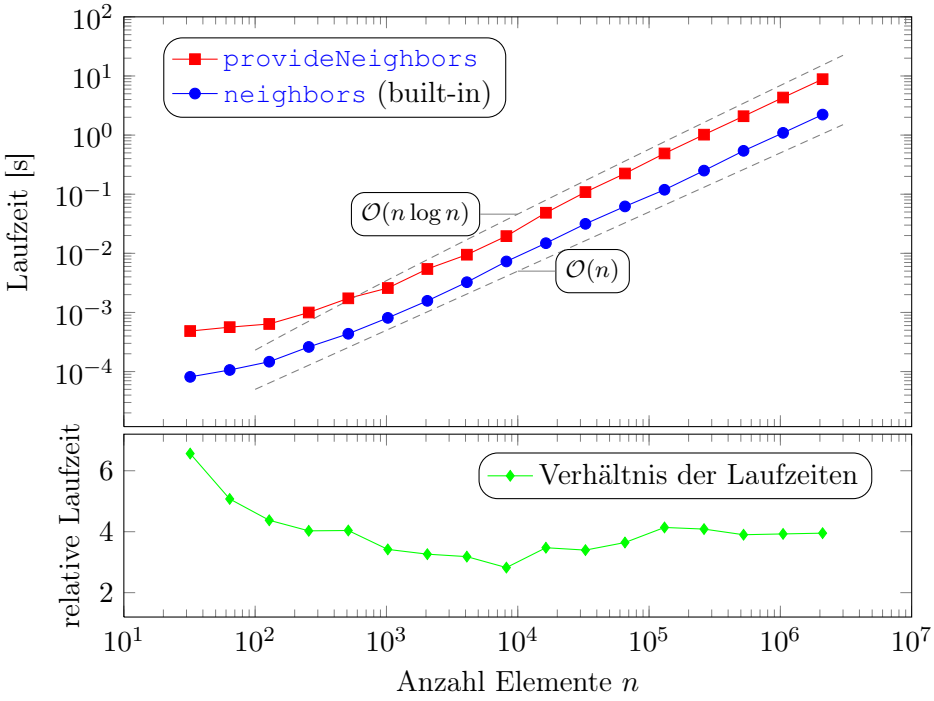


Abbildung 8: Vergleich der Laufzeiten der built-in Funktion `neighbors` mit der besprochenen Funktion `provideNeighbors`. Die built-in Funktion ist in etwa um den Faktor 4 schneller.

```
3  %*** Generierung der Hyperflaechen
4  if size(elements,2)==3
5      dim = 2;
6      hypersurfaceorder = [2,3;1,3;1,2];
7  else
8      dim = 3;
9      hypersurfaceorder = [2,3,4;1,3,4;1,2,4;1,2,3];
10 end
11 %*** Generierung der Hyperflaechen
12 hypersurfaces = sort(reshape(elements(:,hypersurfaceorder),[],dim),2);
13 %*** Finden der zusammengehoerigen Flaechen
14 [~,I,J] = unique(hypersurfaces,'rows');
15 A = I(J);
16 half1 = find(A~=(1:(dim+1)*nE)');
17 half2 = A(half1);
18 %*** Umwandlung der Flaecheninformation in die Nachbarschaftsrelation
19 element2neighbors = zeros(nE,dim+1);
20 if (nE~=1)
21     element2neighbors(half2) = rem(half1-1,nE)+1;
22     element2neighbors(half1) = rem(half2-1,nE)+1;
23 end
```

- Zeile 1: Die Funktion `provideNeighbors` liefert zu einem Array `elements` $\in \mathbb{N}^{n\times 4}$ ein Array `element2neighbors` $\in \mathbb{N}^{n\times 4}$. Der Eintrag `element2neighbors(T,k)` ist dabei die Nummer des Tetraeders, der an den Tetraeder `elements(T,:)` grenzt, aber den Knoten `elements(T,k)` nicht enthält. Falls an der Fläche, die dem Punkt `elements(T,k)` gegenüberliegt kein weiterer Tetraeder liegt, ist der Eintrag `element2neighbors(T,k)`

gleich Null.

- Zeilen 4–10: Abhängig von der übergebenen Triangulierung wird die Dimension der Daten bestimmt. Die angegebene Ordnung liefert in weiterer Folge die gewünschte Eigenschaft `elements(j,k)` $\notin$ `elements(element2neighbors(j,k),:)`.

- Zeile 12: Ein Array `hypersurfaces` wird angelegt, das alle Flächen der Tetraeder enthält. Damit diese eindeutig bestimmt sind, werden die Knoten aufsteigend sortiert. Die Reihenfolge der Flächen in `hypersurfaces` ist dabei wichtig. Die vier Seitenflächen des j-ten Tetraeders `elements(j,:)` sind genau die vier Zeilen

  ```
  hypersurfaces(j+0*nE,:),
  hypersurfaces(j+1*nE,:),
  hypersurfaces(j+2*nE,:) und
  hypersurfaces(j+3*nE,:).
  ```

  Damit kann dann in den Zeilen 19–23 leicht der zur Fläche gehörige Tetraeder bestimmt werden.

- Zeilen 14–17: Hier werden doppelte Flächen gesucht, deren Existenz äquivalent ist zu einem Paar von Nachbarn. Der Befehl `[B,I,J]=unique(F,'rows')` liefert dabei die Indexvektoren `I` und `J`, sodass gilt: `B = F(I,:)` und `F = B(J,:)`, wobei `B` aus `F` entsteht indem mehrfache Vorkommen von Zeilen entfernt werden. Der Vektor `I` enthält dabei immer die Indizes zu den letzten Vorkommen der Zeilen `F(I,:)` in `F`. Dies liefert uns, dass bei doppelten Zeilen die beiden entsprechenden Indizes aus `I` auf das zweite Vorkommen der Zeile zeigen. Der Vektor `A = I(J)` hat somit folgende Einträge:

  - `A(k)==j` mit `j>k`, falls die k-te Fläche zum ersten Mal an der k-ten Stelle, aber insgesamt zumindest doppelt in F vorkommt.
  - `A(k)==k`, falls es das letzte Vorkommen der k-ten Fläche in F ist oder falls die k-te Fläche nicht doppelt vorkommt, es an der entsprechenden Stelle also keinen Nachbarn gibt.

  Damit ist vor allem der erste Fall interessant. Es gilt ja dann schließlich, dass die k-te und j-te Fläche gleich sind, also zwei benachbarten Tetraedern angehören. Diese Indizes j und k entsprechender Flächen werden in die Vektoren `half1` und `half2` gespeichert.

- Zeilen: 19–23 In diesen Zeilen werden zu den sich entsprechenden Paaren von Flächen die zugehörigen Tetraeder gefunden und in die Matrix `element2neighbors` geschrieben. Die Flächen mit Nummern `half1(i)` und `half2(i)` entsprechen einander. Zu einer Flächennummer kann wie bereits erwähnt leicht der zugehörige Tetraeder bestimmt werden. Die Flächennummer modulo der Anzahl der Elemente ergibt die Nummer des zugehörigen Tetraeders, wobei das n-te Element auf die Nummer 0 abgebildet wird. Präziser lässt sich formulieren:

  - Falls gilt `mod(j,nE)==k` mit `k`$\neq 0$, dann gehört die `j`-te Fläche zum `k`-ten Tetraeder.
  - Falls gilt `mod(j,nE)==0`, dann gehört die `j`-te Fläche zum `nE`-ten Tetraeder.

  Diese Fallunterscheidung wird hier indirekt mit dem Befehl `rem(j,nE)` durchgeführt, der den Divisionsrest von `j` durch `nE` liefert. Für `j>=0` gilt also `rem(j,nE)==mod(j,nE)`. Für `j<0` gilt `rem(j,nE)==-mod(-j,nE)`. Damit erhalten wir:

– `rem(j-1,nE)+1==nE`, falls die `j`-te Fläche zum `nE`-ten Tetraeder gehört

– `rem(j-1,nE)+1==mod(j,nE)`, sonst.

Also die gewünschte Eigenschaft. Schließlich erhalten wir das Array `element2neighbors`.

Wir werden in weiteren Verfahren daran interessiert sein, Flächen direkter handhaben zu können. Dazu wird insbesondere eine Nummerierung der Flächen notwendig sein. In Listing 7 generieren wir ebendiese sowie eine Hilfsstruktur, die uns zu gegebenen Tetraedern die zugehörigen Flächen liefert. Die Funktion `provideFaceData` aus Listing 7 liefert zu gegebenen Matrizen `elements` und optionalen Randdaten (in unserem Fall `neumann` und `dirichlet`) die Matrizen `face2nodes`, `element2face` und `varargout`, das für jeden Rand gewissermaßen `boundary2face` darstellt. Die Matrix `face2nodes` beinhaltet zeilenweise alle Flächen $\mathcal{F}$ der Triangulierung. Die Flächen werden dabei im Simplex-Vertex-Format gespeichert. Die $n \times 4$ Matrix `element2faces` liefert Indizes in die Matrix `face2nodes` in der Form, dass die dem Knoten `elements(j,k)` gegenüberliegende Fläche durch `element2faces(j,k)` kodiert wird. Die durch `varargout{:}` repräsentierten Arrays `boundary2face` enthalten Indizes in `element2faces` hinein, sodass die Gleichheit `element2faces(boundary2face(i),:)==boundary(i,:)` gilt.

Listing 7: Geometriedaten für Flächen

```
1  function [face2nodes,element2faces,varargout] = provideFaceData(elements,varargin)
2  nE = size(elements,1);
3  nB = nargin-1;
4  nBE = zeros(1,nB);
5  for j = 1:nB
6      nBE(j) = size(varargin{j},1);
7  end
8  bFInd = cumsum([4*nE,nBE]);
9  %*** Reihenfolge der Flaechen festlegen
10 faceorder = [2,3,4;1,3,4;1,2,4;1,2,3];
11 %*** Knoten der Flaechen aufsteigend anordnen
12 faces = sort(reshape(elements(:,faceorder),[],3),2);
13 %*** Die Randflaechen an faces anhaengen
14 for j = 1:nB
15     boundary = varargin{j};
16     if ~isempty(boundary)
17         faces(bFInd(j)+1:bFInd(j+1),:) = sort(boundary,2);
18     end
19 end
20 %*** Generierung der Flaecheninformationen
21 [face2nodes,~,J] = unique(faces,'rows');
22 element2faces = reshape(J(1:4*nE),[],4);
23 %*** Generierung von boundary2faces
24 for j = 1:nB
25     varargout{j} = reshape(J(bFInd(j)+1:bFInd(j+1)),nBE(j),1);
26 end
```

- Zeile 1: Die Funktion `provideFaceData` liefert zu einem Array `elements`, sowie optionalen Rändern `dirichlet` und `neumann` die Arrays `face2nodes`, `element2faces` sowie `dirichlet2face` und `neumann2face`. Jede Zeile `face2nodes(i,:)` $\in \mathbb{N}^{1\times 3}$ kodiert eine Fläche der Triangulierung. Der Eintrag `j=element2faces(T,k)` liefert den Index `j` der Fläche, die dem Knoten `elements(T,k)` gegenüberliegt. Die Arrays `neumann2face` und `dirichlet2face` erfüllen `face2nodes(neumann2face(F),:)==neumann(F,:)` und entsprechendes für `dirichlet`.

- Zeilen 10–12: Das Verfahren verläuft recht ähnlich der Funktion `provideNeighbors` aus Listing 6. Zuerst wird ein Array generiert, das alle Flächen der Triangulierung zeilenweise enthält. Wieder ist die Nummerierung aus Zeile 10 entscheidend für die richtige Reihenfolge.
- Zeilen 14–19: Um in weiterer Folge die Arrays `boundary2face` zu gewinnen, werden die Randflächen an das Array `faces` angehängt.
- Zeilen 21–26: Die Arrays `element2faces` und `face2nodes` werden mit dem Befehl `unique` generiert. Dazu muss nur `J` in `element2faces` bzw. `varargout` $\hat{=}$ `boundary2faces` aufgeteilt werden.

Schließlich wollen wir ähnliche Datenstrukturen auch für die Kanten der Triangulierung. Diese liefert uns folgende Funktion `provideEdgeData`.

Listing 8: Geometriedaten für Kanten

```
1  function [element2edges,edge2nodes,varargout] = provideEdgeData(elements,varargin)
2  nE = size(elements,1);
3  nB = nargin-1;
4  nBE = zeros(1,nB);
5  for j = 1:nB
6      nBE(j) = size(varargin{j},1);
7  end
8  bEInd = cumsum([nE*6,3*nBE]);
9  %*** Reihenfolge der Kanten festlegen
10 edgeorder = [1,2;1,3;1,4;2,3;2,4;3,4];
11 boundaryedgeorder = [2,3;1,3;1,2];
12 %*** Knoten der Kanten aufsteigend sortieren
13 edges = sort(reshape(elements(:,edgeorder),6*nE,2),2);
14 %*** Die Kanten der Randflaechen an edges anhaengen
15 for j=1:nB
16     boundary = varargin{j};
17     if ~isempty(boundary)
18         edges(bEInd(j)+1:bEInd(j+1),:) = ...
19                 sort(reshape(boundary(:,boundaryedgeorder),[],2),2);
20     end
21 end
22 %*** Generierung der Kanteninformation
23 [edge2nodes,~,J] = unique(edges,'rows');
24 element2edges = reshape(J(1:nE*6),nE,6);
25 %*** Generierung von boundary2edges
26 for j = 1:nB
27     varargout{j} = reshape(J(bEInd(j)+1:bEInd(j+1)),nBE(j),3);
28 end
```

- Zeile 1: Die Funktion `provideEdgeData` liefert zu einem Array `elements`$\in \mathbb{N}^{n\times 4}$, sowie optionalen Rändern `dirichlet` und `neumann` die Arrays `element2edges`, `edge2nodes` sowie `dirichlet2edges` und `neumann2edges`. Jede Zeile `edge2nodes(i,:)` $\in \mathbb{N}^{1\times 2}$ kodiert eine Kante der Triangulierung. Die Reihenfolge der sechs Kanten je Tetraeder in `element2edges` $\in \mathbb{N}^{n\times 6}$ ist in Zeile 10 gegeben. Die Reihenfolge der Kanten einer Randfläche ist in Zeile 11 gegeben. Sie ist so gewählt, dass `edge2nodes(neumann2edges(F,k),:)` eine Kante der Fläche `neumann(F,:)` ist und dem Punkt `neumann(F,k)` gegenüberliegt. Gleiches gilt für `dirichlet2edges`.

- Zeilen 10–21: Gänzlich analog zu Listing 7 wird hier das Arrays `edges` erstellt. Dabei wird bei der Reihenfolge der Kanten der Tetraeder als Konvention die „lexikographische Ordnung“ aus Zeile 10 eingeführt. Bei den Kanten der Randflächen gilt ähnlich wie zuvor: `boundary2edges(j,k)` ist die Kante, die gegenüber vom Knoten `elements(j,k)` liegt.
- Zeilen 23–28: Mittels `unique` erreichen wir wie in Listing 7 das gewünschte Ziel.

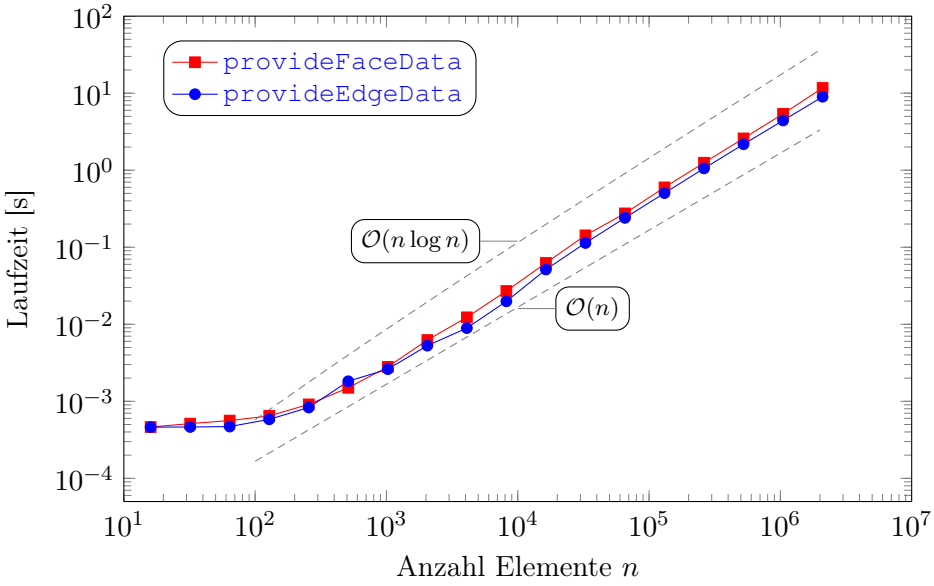


Abbildung 9: Laufzeiten der Funktionen `provideEdgeData` und `provideFaceData`. Den Erwartungen gemäß skalieren beide fastlinear in Abhängigkeit der Anzahl der Elemente.

Die Laufzeiten der beiden Funktionen `povideEdgeData` und `provideFaceData` verhalten sich wie $\mathcal{O}(n \log n)$ in Abhängigkeit der Anzahl Elemente n, wie in Abbildung 9 ersichtlich ist.

5.2 Eine uniforme Verfeinerung

Als erstes Verfeinerungsschema werden wir ein uniformes besprechen, bei dem also alle Tetraeder der Triangulierung verfeinert werden. Wir stellen die Verallgemeinerung des als *Rotverfeinerung* bekannten Verfahrens auf den dreidimensionalen Fall vor, bei dem sämtliche Kanten der Triangulierung halbiert werden. Solche Verfahren nennt man unabhängig von der Dimension – und im Namenskonflikt zu unserem Triangulierungsbegriff – *reguläre Verfeinerungen*. Das Verfahren funktioniert nun so: Jeder Tetraeder wird dabei in acht kleinere Tetraeder zerlegt. Anschaulich werden zuerst alle vier Ecken jedes Tetraeders abgeschnitten, wodurch jeweils vier neue Tetraeder entstehen. Die übrigbleibenden Oktaeder werden dann in jeweils vier Tetraeder unterteilt. Das Aufteilen der Oktaeder in die Tetraeder ist dabei nicht eindeutig, sondern hängt von der Wahl der Diagonalen ab entlang der aufgeteilt werden soll. Zhang [20] konnte dazu zeigen, dass bei Wahl der kürzesten Diagonalen die kleinsten Innenwinkel der entstehenden Tetraeder für die Folge der Verfeinerungen beschränkt bleibt. Eine andere Herangehensweise ist die von Bey [4], der die Wahl der Diagonale abhängig von einer anfangs gewählten Nummerierung der Knoten macht. Dies nennt man ein *typographisches* Verfahren. Wir werden nun dieses Verfahren schildern und eine entsprechende Implementierung in MATLAB besprechen. Wir wollen in diesem Abschnitt ein Tetraeder T mit einem geordneten Tupel seiner Knoten $T = (v_1, v_2, v_3, v_4)$

identifizieren. Mit $v_{ij} := (v_i + v_j)/2$ für $0 \leq i < j \leq 4$ sei der Halbierungspunkt der Kante zwischen v_i und v_j bezeichnet.

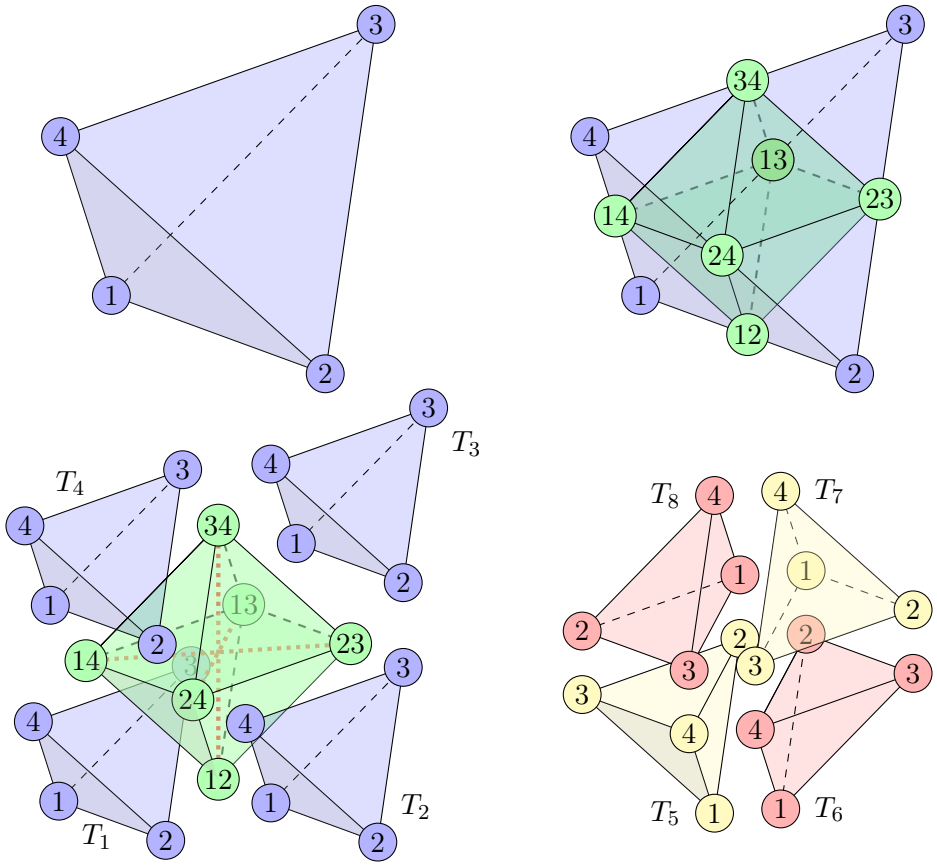


Abbildung 10: Visualisierung der Rotverfeinerung: Indem alle Kanten des Tetraeders halbiert werden entstehen vier neue Tetraeder, die ähnlich zum ursprünglichen sind. Der verbleibende Oktaeder könnte auf drei verschiedene Arten (rot gepunktet) in vier Tetraeder unterteilt werden. Die hier dargestellte Variante, nach Bey [4] zeigt wie die Knoten der neu entstehenden Elemente zu nummerieren sind. Die Ziffern 1–4 sind jeweils als tetraederinterne Nummerierung der Knoten zu verstehen.

Satz 5.2.1 (Bey, [4]). *Das Verfeinerungsschema, das einem Tetraeder $T = (v_1, v_2, v_3, v_4)$ die acht Tetraeder*

$$\begin{aligned} T_1 &:= (v_1,\ v_{12}, v_{13}, v_{14}), & T_5 &:= (v_{12}, v_{13}, v_{14}, v_{24}) \\ T_2 &:= (v_{12}, v_2,\ v_{23}, v_{24}), & T_6 &:= (v_{12}, v_{13}, v_{23}, v_{24}) \\ T_3 &:= (v_{13}, v_{23}, v_3,\ v_{34}), & T_7 &:= (v_{13}, v_{23}, v_{24}, v_{34}) \\ T_4 &:= (v_{14}, v_{24}, v_{34}, v_4\ ), & T_8 &:= (v_{13}, v_{14}, v_{24}, v_{34}) \end{aligned}$$

zuweist, liefert rekursiv angewandt reguläre Triangulierungen von T. Alle entstehenden Tetraeder liegen dabei in höchstens drei verschiedenen Ähnlichkeitsklassen.

Das Verfeinerungsschema wird in Abbildung 10 visualisiert. Erstmals ist nun die Reihenfolge der Knoten in der Datenstruktur `elements` von Bedeutung. Ein Umordnen der Knoten zwischen einzelnen Verfeinerungsschritten führt zweifelsohne dazu, dass die Eigenschaft aus Satz

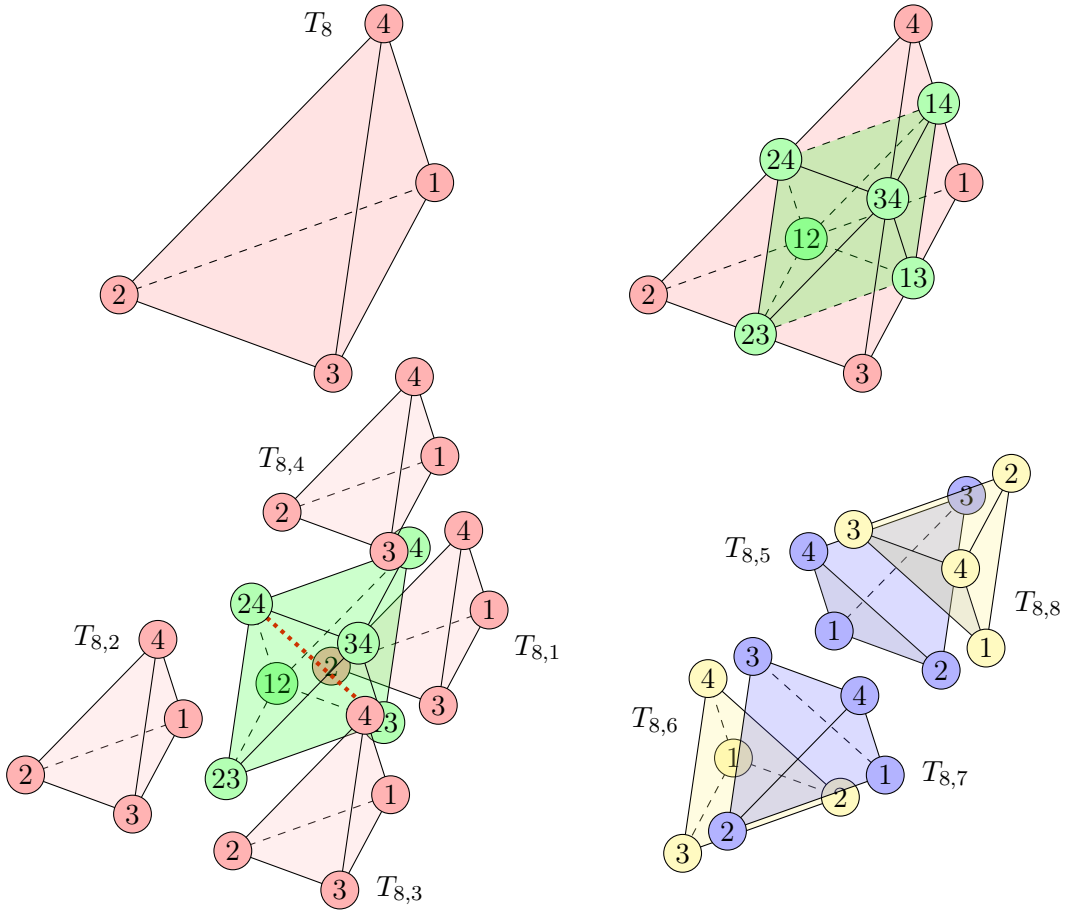


Abbildung 11: Visualisierung der Rotverfeinerung: Weitere Verfeinerung des neu entstandenen Tetraeders T_8. Dabei entstehen keine zusätzlichen Ähnlichkeitsklassen.

5.2.1 nicht mehr garantiert werden kann. Satz 5.2.1 macht zwar keine Aussage über die Verfeinerung von ganzen Triangulierungen, man sieht allerdings, dass aneinanderliegende Flächen (sogar alle Flächen) gleich verfeinert werden wodurch weder hängende Knoten, noch Überlappungen entstehen können. Gleichzeitige Unterteilung einer gesamten regulären Triangulierung liefert also wieder eine gültige Triangulierung. Nun zur Implementierung in MATLAB.

Listing 9: ROTVERFEINERUNG

```
function [elements,coordinates,varargout] ...
                                    = redRefine(elements,coordinates,varargin)
nB = nargin-2;
nC = size(coordinates,1);
[edge2nodes,element2edges,boundary2edges{1:nB}] ...
                                    = provideEdgeData(elements,varargin{1:nB});
%*** Erzeuge neue Knoten
coordinates = [coordinates;0.5*( coordinates(edge2nodes(:,1),:) ...
                                +coordinates(edge2nodes(:,2),:) )];
%*** Vektorisierte Verfeinerung der einzelnen Tetraeder
elements = [elements(:,1),nC+element2edges(:,[1,2,3]);...
            nC+element2edges(:,[1]),    elements(:,2),nC+element2edges(:,[4,5]);...
            nC+element2edges(:,[2,4]),  elements(:,3),nC+element2edges(:,[6]  );...
```

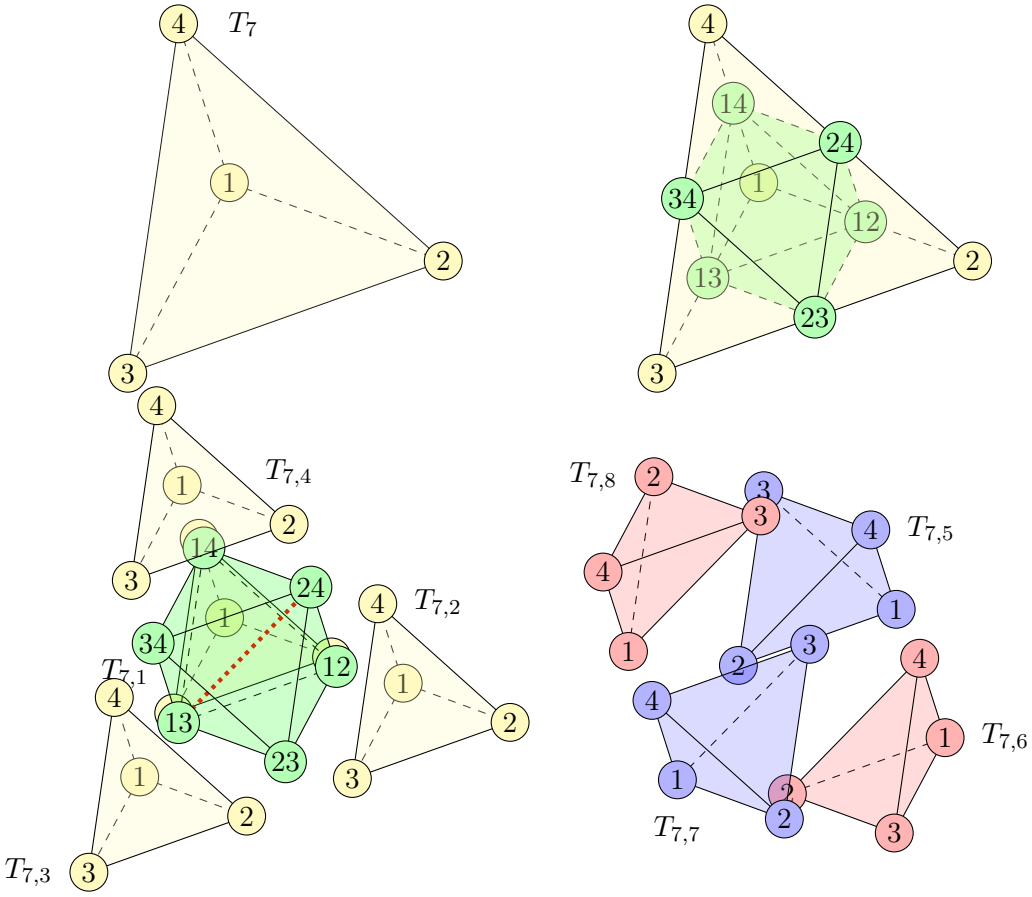


Abbildung 12: Visualisierung der Rotverfeinerung: Weitere Verfeinerung des neu entstandenen Tetraeders T_7. Dabei entstehen keine zusätzlichen Ähnlichkeitsklassen.

```
14              nC+element2edges(:,[3,5,6]),elements(:,4);...
15              nC+element2edges(:,[1,2,3,5]);...
16              nC+element2edges(:,[1,2,4,5]);...
17              nC+element2edges(:,[2,4,5,6]);...
18              nC+element2edges(:,[2,3,5,6]);...
19              ];
20  %*** Vektorisierte Verfeinerung der Randflaechen
21  for j=1:min(nargout-2,nB)
22      if ~isempty(varargin{j})
23          varargout{j} = [varargin{j}(:,1),nC+boundary2edges{j}(:,[3,2]);...
24                          varargin{j}(:,2),nC+boundary2edges{j}(:,[1,3]);...
25                          varargin{j}(:,3),nC+boundary2edges{j}(:,[2,1]);...
26                          nC+boundary2edges{j}(:,[1,2,3])];
27      else
28          varargout{j} = [];
29      end
30  end
```

- Zeile 1: Die Funktion `redRefine` übernimmt eine reguläre Triangulierung $\mathcal{T}$ in Form von Arrays `elements` und `coordinates`, sowie optionale Arrays `neumann` und `dirichlet`. Sie führt eine uniforme Rotverfeinerung durch und gibt die resultierende Triangulierung zurück.

- Zeile 8–9: Die neuen Knoten werden generiert. Zusätzlich zu den alten Knoten kommen noch für jede Kante die Halbierungspunkte dazu.
- Zeilen 11–19: Hier werden die neuen Tetraeder gemäß dem Verfeinerungsschema aus Satz 5.2.1 erzeugt. Weil alle Kanten verfeinert werden ist `nC+element2edges(j,k)` der Index des neuen Knoten, der durch Halbieren der Kante `element2edges(j,k)` entsteht.
- Zeilen 21–30: Die Randflächen werden allesamt mittels einer zweidimensionalen Rotverfeinerung verfeinert. Damit passen die Randflächen wieder mit den neuen Tetraedern zusammen.

Wir haben nun die Möglichkeit, unser Modell mit immer feiner werdenden Triangulierungen des Gebietes zu berechnen. Dies natürlich nur, wenn man von den Limitierungen der verfügbaren Rechen- und Speicherkapazität absieht. Angesichts dieser Einschränkungen stellt sich nun die Frage, ob es nicht angemessener wäre, nur dort von der Verfeinerung Gebrauch zu machen, wo sie am dringendsten benötigt wird. Für solch ein adaptives Verfahren ist die Rotverfeinerung unbrauchbar, da sie bei nicht-globaler Anwendung hängende Knoten produzieren würde. Wir wenden uns nun also einem anderen Verfahren zu.

5.3 Ein Bisektionsverfahren

Eine andere Klasse von Verfeinerungen sind die sogenannten *Bisektionsverfahren*. Diese stellen die sukzessive Halbierung der Tetraeder der Starttriangulierung dar. Hierbei geht man so vor, dass ein gegebener Tetraeder entlang der Hyperebene halbiert wird, die vom Mittelpunkt einer ausgewählten Kante und den gegenüberliegenden Knoten aufgespannt wird. Ähnlich wie bei oben genannter Verfeinerung ist die Bisektion nicht eindeutig definiert, da nicht klar ist welche Kante halbiert werden soll. Auch hier gibt es wieder den Ansatz eines geometrischen (longest edge bisection) nach Rivara [13] und eines *typographischen* Schemas nach Maubach [9] oder Traxler [15], die auch im allgemeinen d-dimensionalen Fall anwendbar sind. Für alle diese Verfahren konnte gezeigt werden, dass der kleinste Innenwinkel der entstehenden Tetraeder beschränkt ist und sie damit nicht zu sehr entarten. Für das Schema aus [15] zeigte Bey in [3], dass für einen d-dimensionalen Simplex die Anzahl der durch Bisektion entstehenden Ähnlichkeitsklassen durch $d!2^{d-2}d$ beschränkt ist, wobei diese Schranke im Allgemeinen sogar scharf ist. Ebendieses Verfahren wurde von Stevenson in [14] aufgegriffen und in den Bedingungen, die an die Starttriangulierung gestellt werden, abgeschwächt. Das hat allerdings zur Folge, dass der Bisektionsalgorithmus von Traxler nur mehr in modifizierter Form anwendbar ist, da sich die Definitionen der *gespiegelten Nachbarn* bei diesen Autoren nicht decken. Wir werden den dreidimensionalen Spezialfall von [14] als Grundlage für diesen Abschnitt verwenden. Wie bei der Rotverfeinerung ist uns jetzt die Reihenfolgen der Knoten wichtig:

Definition 5.3.1. *Für einen nicht entarteten Tetraeder* $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ *unterscheiden wir zwischen* $3 \cdot 4!$ *geordneten Tupeln*

$$\big(v_{\pi(1)}, v_{\pi(2)}, v_{\pi(3)}, v_{\pi(4)}\big)_\gamma \qquad \textit{mit } \gamma \in \{0,1,2\} \textit{ und } \pi \textit{ Permutation von } \{1,2,3,4\}$$

die wir jeweils als markierter Tetraeder vom Typ γ *bezeichnen. Den Triangulierungsbegriff aus Definition 1.2.2 übernehmen wir diesen neuen Repräsentanten eines Tetraeders entsprechend.*

Das folgende Schema ist der zentrale Baustein für das Verfahren dieses Abschnitts.

Definition 5.3.2. *Für einen markierten Tetraeder* $T = (v_1, v_2, v_3, v_4)_\gamma$ *definieren wir die beiden Kinder*

$$T_1 = (v_1, v_{14}, v_2, v_3)_{(\gamma+1) \bmod 3} \quad \textit{und} \quad T_4 = \begin{cases} (v_4, v_{14}, v_2, v_3)_{(\gamma+1) \bmod 3} & \textit{falls } \gamma \in \{1, 2\}, \\ (v_4, v_{14}, v_3, v_2)_{(\gamma+1) \bmod 3} & \textit{falls } \gamma = 0. \end{cases}$$

Wobei $v_{14} := (v_1 + v_4)/2$ *der Halbierungspunkt der Kante zwischen* v_1 *und* v_4 *ist. Wir werden in Anlehnung an die Speicherung als Binärbaum* T_1 *als* linkes Kind *und entsprechend* T_4 *als* rechtes Kind *bezeichnen.*

Die Kinder entstehen also durch Halbieren der Kante $\operatorname{conv}\{v_1, v_4\}$ – die wir deshalb als *Referenzkante* bezeichnen werden – und anschließendem Verbinden mit den gegenüberliegenden Knoten, sowie geeigneter Umordnung dieser (siehe Abbildung 13). Dieses Schema liefert für einen Tetraeder höchstens $d!2^{d-2}d = 36$ verschiedene Ähnlichkeitsklassen (Siehe [3]). Damit haben wir schon eine wichtige Eigenschaft für ein brauchbares Verfeinerungsverfahren. Nun erläutern wir, wie man aus der Verfeinerungsregel aus Definition 5.3.2 ein vollständiges Verfahren erhält.

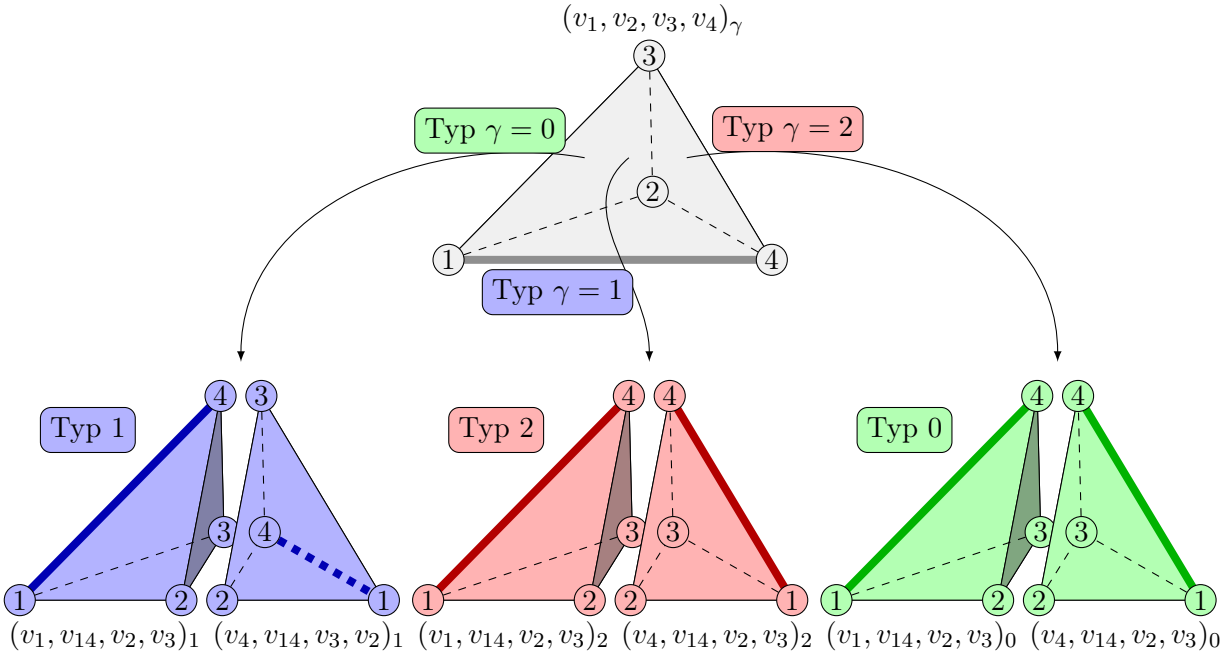


Abbildung 13: Das Verfeinerungsschema aus Definition 5.3.2. Die fett markierten Kanten sind die Referenzkanten der Tetraeder. Die Ziffern 1–4 sind wieder die Positionen der Knoten im entsprechenden Tupel.

Definition 5.3.3. *Für einen markierten Tetraeder* $T = (v_1, v_2, v_3, v_4)_\gamma$ *definieren wir*

$$T_R := \begin{cases} (v_4, v_2, v_3, v_1)_\gamma, & \textit{falls } \gamma \in \{1, 2\}, \\ (v_4, v_3, v_2, v_1)_\gamma, & \textit{falls } \gamma = 0. \end{cases}$$

Aus der Verfeinerung geht hervor, dass T_R *der einzige andere Tetraeder vom Typ* γ *ist, der auch die Kinder* T_1 *und* T_4 *erzeugt. Weiters nennen wir einen markierten Tetraeder* T' *einen* gespiegelten Nachbarn *von* T*, wenn sich eines der Tupel* T *oder* T_R *nur an einer Stelle von* T' *unterscheidet.*

Definition 5.3.4. *Wir nennen eine Triangulierung $\mathcal{T}_0$ eine* passende Starttriangulierung *falls gilt:*

- *$\mathcal{T}_0$ ist regulär,*
- *alle Tetraeder $T \in \mathcal{T}_0$ haben den gleichen Typ γ,*
- *für je zwei benachbarte Tetraeder $T = (v_1, \ldots, v_4)_\gamma$, $T' = (v'_1, \ldots, v'_4)_\gamma$ aus $\mathcal{T}_0$ gelten die Bedingungen:*
  - *Falls eine der Referenzkanten $\operatorname{conv}\{v_1, v_4\}$ oder $\operatorname{conv}\{v'_1, v'_4\}$ in $T \cap T'$ liegt, dann ist T sogar gespiegelter Nachbar von T'.*
  - *Falls keine der Refrenzkanten in $T \cap T'$ liegt, dann sind die benachbarten Kinder von T und T' gespiegelte Nachbarn.*

Für passende Starttriangulierungen lassen sich nun zwei Sätze zeigen, deren Gültigkeit das Verfeinerungsverfahren dieses Abschnitts liefert.

Definition 5.3.5. *Die Generation eines Tetraeders definieren wir rekursiv durch: $\ell(T_0) = 0$ für alle $T_0 \in \mathcal{T}_0$ und $\ell(T') = \ell(T) + 1$ für ein Kind T' von T. Weiters definieren wir eine Menge von Tetraedern als* kompatibel teilbar *wenn alle dieselbe Referenzkante haben. Wenn alle Tetraeder einer Kante kompatibel teilbar sind, dann bleibt durch deren Bisektion insbesondere die Regularität der Triangulierung erhalten. Schließlich definieren wir noch die Menge*

$$N(\mathcal{T}, T) := \{T' \in \mathcal{T} : T' \text{ ist Nachbar von } T \text{ und enthält die Referenzkante von } T\}.$$

Satz 5.3.6 (Stevenson [14], Abschnitt 4)**.** *Jede uniforme Verfeinerung von $\mathcal{T}_0$ – das ist eine durch Bisektion aus $\mathcal{T}_0$ entstandene Triangulierung, deren Tetraeder alle dieselbe Generation haben – ist regulär.*

Dieser Satz ist natürlich für sich genommen bereits interessant, da sich damit ein uniformes Verfeinerungsverfahren erzeugen ließe (vgl. Abbildung 14), es lässt sich damit aber auch folgender Satz beweisen:

Satz 5.3.7 (Stevenson [14], Abschnitt 4)**.** *Jede reguläre Triangulierung $\mathcal{T}$ die man durch Anwendung des Verfeinerungsschemas der Definition 5.3.2 aus einer passenden Starttriangulierung gewinnt erfüllt, dass für alle $T \in \mathcal{T}$ und $T' \in N(\mathcal{T}, T)$ gilt:*

- *Entweder $\ell(T') = \ell(T)$ und T ist kompatibel mit T' teilbar,*
- *oder $\ell(T') = \ell(T) - 1$ und T ist kompatibel mit einem der Kinder von T' teilbar.*

Von diesem Satz macht nun direkt Listing 10 Gebrauch, das [14] entnommen ist. Zu einer gegebenen Triangulierung $\mathcal{T}$ und einem Tetraeder T liefert sie die gröbste reguläre Verfeinerung von $\mathcal{T}$, in der T geteilt wurde.

Listing 10: Refine

```
function T' = refine[T, T]
K := ∅; F := {T}
do Fnew := ∅
    forall T' ∈ F do
        forall T'' ∈ N(T, T') with T'' ∉ F ∪ K do
            if T'' kompatibel mit T' teilbar
            then Fnew := Fnew ∪ {T''}
```

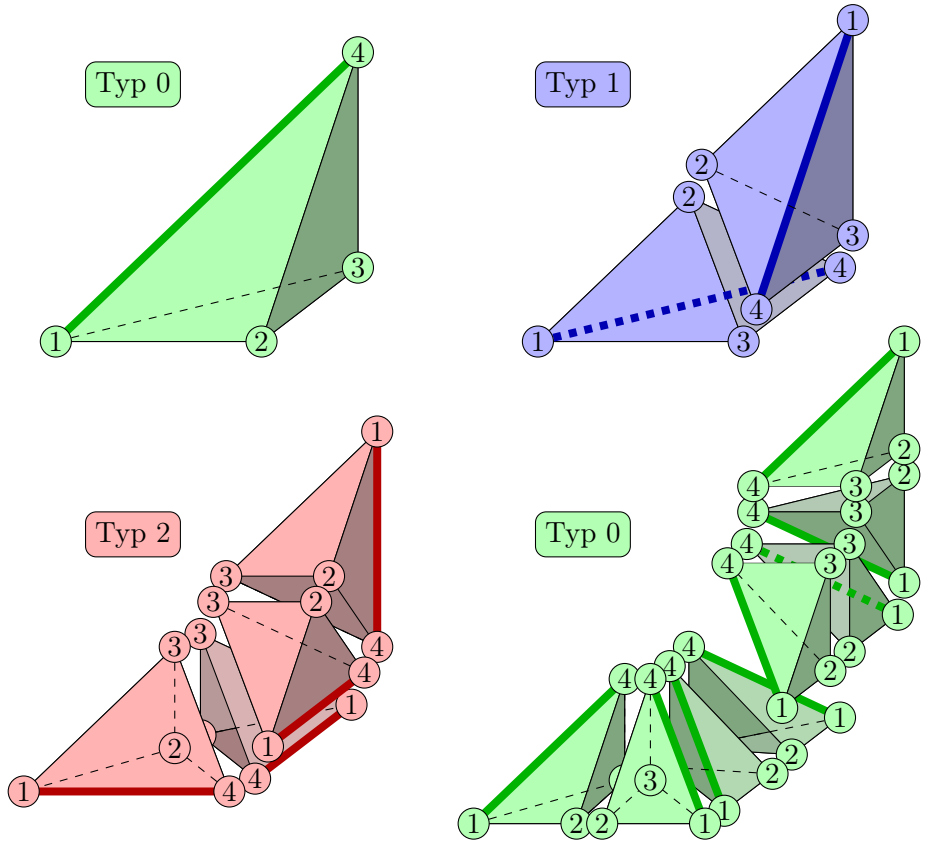


Abbildung 14: Uniforme Verfeinerungen mittels Bisektion. Diese sind nach Satz 5.3.6 also insbesondere regulär. Die fett gezeichneten Kanten sind die Verfeinerungskanten der jeweiligen Tetraeder.

```
            else P :=refine[T, T'']
                Fuege zu Fnew das Kind von T'' hinzu, das Nachbar von T' ist
            endif
        endfor
    endfor
    K := K ∪ F
    F := Fnew
until F = ∅
Erstelle T' aus T durch gleichzeitige Bisektion aller Elemente aus K
```

Anschaulich beginnt die Funktion bei T und überprüft, ob alle Tetraeder, die adjazent zur Verfeinerungskante von T sind, auch kompatibel teilbar sind. Falls dies der Fall ist, werden alle diese Tetraeder halbiert, und die Funktion terminiert. Falls allerdings einer dieser Tetraeder nicht kompatibel mit T teilbar sein sollte, dann folgt aus Satz 5.3.7, dass dieser lediglich halbiert werden muss, um ein kompatibel teilbares Kind zu liefern. So versucht also die Funktion zuerst, den benachbarten Tetraeder zu teilen, wobei bei diesem auch überprüft werden muss ob alle zugehörigen Tetraeder kompatibel teilbar sind. Das Verfahren führt also genau alle zur Erhaltung der Regularität notwendigen Bisektionsschritte durch und zwar in einer Reihenfolge, durch die in jedem Schritt eine reguläre Triangulierung vorliegt. Nun stellt sich nur noch die Frage wie man zu einer gegebenen Triangulierung eine passende Starttriangulierung erhalten kann. Die Antwort darauf liefert folgendes Verfahren:

Satz 5.3.8 (Kossaczký [7] und Stevenson [14] für $n \geq 3$). *Sei $\mathcal{T}$ eine reguläre Triangulierung. Für jeden Tetraeder $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\} \in \mathcal{T}$ seien zwölf markierte Tetraeder definiert durch:*

$$
\begin{aligned}
T_{11} &:= (v_2, S_1, S, v_3)_2, & T_{31} &:= (v_1, S_3, S, v_2)_2,\\
T_{12} &:= (v_2, S_1, S, v_4)_2, & T_{32} &:= (v_1, S_3, S, v_4)_2,\\
T_{13} &:= (v_3, S_1, S, v_4)_2, & T_{33} &:= (v_2, S_3, S, v_4)_2,\\
\\
T_{21} &:= (v_1, S_2, S, v_3)_2, & T_{41} &:= (v_1, S_4, S, v_2)_2,\\
T_{22} &:= (v_1, S_2, S, v_4)_2, & T_{42} &:= (v_1, S_4, S, v_3)_2,\\
T_{23} &:= (v_3, S_2, S, v_4)_2, & T_{43} &:= (v_2, S_4, S, v_3)_2.
\end{aligned}
$$

Dabei sei S der Schwerpunkt des Tetraeders und S_i der Schwerpunkt der Seitenfläche, die dem Knoten v_i gegenüberliegt. Dann ist die Menge aller so entstandenen Tetraeder eine passende Starttriangulierung.

In etwas anschaulichere Worte gefasst, lautet die Regel aus obigem Satz so: Zuerst wird T in vier Tetraeder zerteilt, indem wir im Schwerpunkt S einen neuen Knoten einführen. Im nächsten Schritt wird bei jedem dieser vier Tetraeder an der Fläche, die er mit T gemeinsam hat der Schwerpunkt eingeführt und der Tetraeder anhand dieses neuen Knotens in jeweils drei neue Tetraeder zerlegt. Nun wird die Markierung der Tetraeder so gewählt, dass nur die ursprünglichen Kanten von T Referenzkanten sind, und die Position des Schwerpunkts S immer an dritter Stelle liegt. Dass die Bedingungen aus Definition 5.3.4 erfüllt sind, lässt sich innerhalb eines Tetraeders leicht überprüfen und folgt für benachbarte Tetraeder aus der Symmetrie der Markierungen bezüglich der baryzentrischen Koordinaten. Beachte, dass die Verfeinerungsregel zwar abhängig von der Repräsentation der ursprünglichen Tetraeder ist, da man den Tetraeder $T = \operatorname{conv}\{v_1, v_2, v_3, v_4\}$ auf 4! verschiedene Arten als markierten Tetraeder darstellen kann, der Satz jedoch unabhängig von der ursprünglichen Knotenreihenfolge gilt.

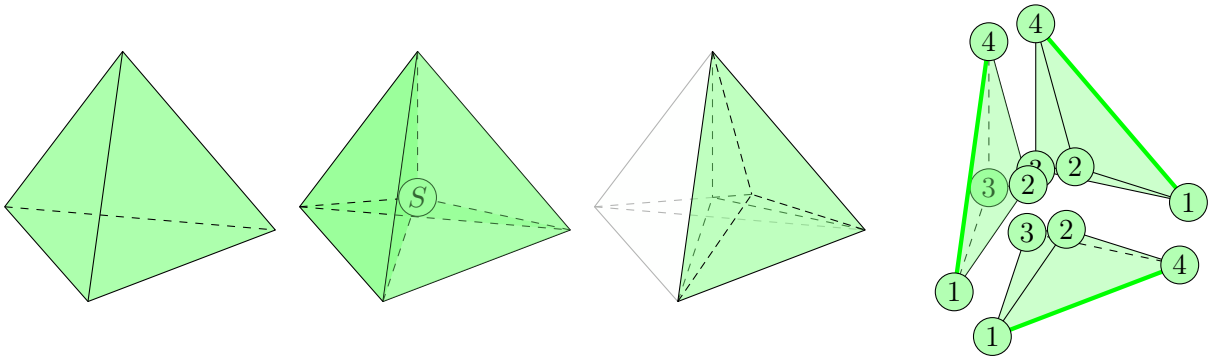


Abbildung 15: Verfahren zur Generierung einer passenden Starttriangulierung. Drei der zwölf entstehenden Tetraeder sind dargestellt. Alle Schwerpunkte der ursprünglichen Elemente stehen in den resultierenden Tetraedern an dritter, alle Schwerpunkte der Randflächen an zweiter Stelle. Die Referenzkanten liegen an den Kanten der ursprünglichen Triangulierung.

Bevor wir nun erst eine MATLAB-Implementierung des Verfahrens aus Satz 5.3.8 und anschließend des Bisektionsverfahrens aus Listing 10 besprechen, sei noch ein Ergebnis und eine daraus

resultierende Konvention vorangestellt. In [17] wurde gezeigt, dass bei Anwendung des von uns verwendeten Bisektionsverfahrens auf einen d-dimensionalen Simplex, dessen Hyperflächen (und induktiv auch deren Hyperflächen) nach demselben Bisektionsverfahren für die entsprechend niedrigere Dimension geteilt werden. In unserem Spezialfall bedeutet das, dass die Randflächen eines Tetraeders entsprechend mittels *newest vertex bisection* geteilt werden. Dies begründet nun unsere Wahl der Speicherung einer Randfläche $\operatorname{conv}\{v_1, v_2, v_3\}$ in der Form `[1,2,3]`, wobei die Kante $\operatorname{conv}\{v_1, v_3\}$ die Referenzkante und der Knoten v_2 den neuesten Knoten des Dreiecks darstellt.

Für die Datenstruktur der Triangulierung setzen wir zur Anwendung des Bisektionsverfahrens fest:

Definition 5.3.9. *Wir setzen fest:*

- *Die Reihenfolge der Einträge des $M \times 4$ Arrays* `elements` *sind nun auch von Bedeutung. Wir speichern den markierten Tetraeder $T_\ell = (v_i, v_j, v_k, v_m)_\gamma$ als*

  $$\texttt{elements(}\ell\texttt{,:) = [i,j,k,m].}$$

  Damit sind `elements(:,[1,4])` *stets die Referenzkanten und* `elements(:,2)` *die neuesten Knoten der Tetraeder.*

- *Anstatt des Typs speichern wir die Generation (also die Anzahl der angewandten Bisektionen) des Tetraeders im $M \times 1$ Array* `elementgeneration`*, sodass damit gilt*

  $$\texttt{mod(elementgeneration(}\ell\texttt{),3) == }\gamma.$$

- *Wir speichern die Arrays* `dirichlet` *und* `neumann` *so, dass die Kanten* `dirichlet(:,[1,3])` *und* `neumann(:,[1,3])` *stets die Referenzkanten der Randflächen sind. Darauf muss – wie zuvor bemerkt in [17] gezeigt wurde – vor allem bei der Starttriangulierung geachtet werden, es folgt durch korrekt angewandte zweidimensionale Bisektion sodann automatisch.*

- *Die durch Bisektion entstandenen Kinder von $T = (v_1, v_2, v_3, v_4)_\gamma$ werden stets so gespeichert, dass das Kind, das den Knoten v_1 enthält* (linkes Kind) *im Array* `elements` *vor dem anderen Kind* (rechtes Kind) *und dessen weiteren Kindern steht. (Dafür gibt es beispielsweise die Möglichkeiten das linke Kind an Stelle des Elternelements zu speichern und das rechte Kind an das Ende des Arrays zu hängen, oder die – in* `refineC` *verwendete – Variante die beiden Kinder in richtiger Reihenfolge an der Position des Elternelements einzufügen.)*

- *Genauso stehe in den Arrays* `neumann` *und* `dirichlet` *das linke Kind der Randfläche $F = (v_1, v_2, v_3)$ stets vor dem rechten Kind.*

Nun folgt der Code zur Generierung einer passenden Starttriangulierung.

Listing 11: validStartMesh

```
1  function [elements,elementgeneration,coordinates,varargout] = ...
2                                  validStartMesh(elements,coordinates,varargin)
3  nE = size(elements,1);
4  nC = size(coordinates,1);
5  nB = nargin-2;
6  nBE = zeros(1,nB);
7  for j = 1:nB
```

```
8     nBE(j) = size(varargin{j},1);
9  end
10 %*** Erzeugen der Flaecheninformationen
11 [face2nodes,element2faces,boundary2faces{1:nB}] = ...
12                               provideFaceData(elements,varargin{:});
13 nF = size(face2nodes,1);
14 %*** Generierung der Schwerpunkte
15 elementcenter = 1/4*(  coordinates(elements(:,1),:) ...
16                       + coordinates(elements(:,2),:) ...
17                       + coordinates(elements(:,3),:) ...
18                       + coordinates(elements(:,4),:) );
19 facecenter = 1/3*(  coordinates(face2nodes(:,1),:) ...
20                    + coordinates(face2nodes(:,2),:) ...
21                    + coordinates(face2nodes(:,3),:) );
22 coordinates = [coordinates;elementcenter;facecenter];
23 %*** Bestimmung der neuen Knotennummern
24 element2newNodes = nC+(1:nE)';
25 face2newNodes = nE+nC+(1:nF)';
26 %*** Erstellung der neuen Elemente
27 elements = [face2nodes(element2faces,1),face2newNodes(element2faces,:),...
28                   repmat(element2newNodes,4,1),face2nodes(element2faces,2);...
29              face2nodes(element2faces,1),face2newNodes(element2faces,:),...
30                   repmat(element2newNodes,4,1),face2nodes(element2faces,3);...
31              face2nodes(element2faces,2),face2newNodes(element2faces,:),...
32                   repmat(element2newNodes,4,1),face2nodes(element2faces,3) ];
33 elementgeneration = 2*ones(12*nE,1);
34 %*** Erstellung der Randflaechen
35 for j = 1:min(nargout-2,nB)
36    if ~isempty(varargin{j})
37       varargout{j} = [face2nodes(boundary2faces{j},1),...
38                       face2newNodes(boundary2faces{j}),...
39                       face2nodes(boundary2faces{j},2);...
40                       face2nodes(boundary2faces{j},1),...
41                       face2newNodes(boundary2faces{j}),...
42                       face2nodes(boundary2faces{j},3);...
43                       face2nodes(boundary2faces{j},2),...
44                       face2newNodes(boundary2faces{j}),...
45                       face2nodes(boundary2faces{j},3) ];
46    end
47 end
```

- Zeile 1: Die Funktion übernimmt eine reguläre Triangulierung in Form der Arrays `elements`, `coordinates` und optionalen Randflächen `neumann`, `dirichlet`. Sie liefert eine für die Bisektion passende Starttriangulierung im Sinne von Definition 5.3.4. Dabei gibt das Array `elementgeneration` sowohl Auskunft über die Generation, als auch über den Typ der Tetraeder, und zwar durch die Gleichheit

  ```
  mod(elementgeneration,3)==elementtype.
  ```

- Zeilen 14–25: Hier werden die Schwerpunkte der Flächen, sowie Tetraeder bestimmt und dem Array `coordinates` hinzugefügt. Auch werden die Arrays `element2newNodes` und `face2newNodes` aufgebaut, die die Zuordnung für den Aufbau der Elemente liefert.

- Zeilen 27–33: Die Elemente werden gemäß Satz 5.3.8 bestimmt. Dazu sei die Reihenfolge der Flächen

  ```
  faceorder = [2,3,4; 1,3,4; 1,2,4; 1,2,3];
  ```

in Erinnerung gerufen, welche die Formel für `elements` liefert. Wir werden nicht nur die Typen der Elemente speichern, sondern sogar deren Generation. Dies geschieht in Hinblick auf das Vergröbern einer Triangulierung. Durch Speicherung der Generation ist somit leichter überprüfbar ob ein Tetraeder vergröbert werden darf, da es durch die Funktion `validStartMesh` zu einer notwendigen Voraussetzung wird, dass die Elementgeneration (wegen der Wahl der Generation in Satz 5.3.8) größer als zwei ist.

- Zeilen 35–47: Die Randflächen werden in passender Form für das zweidimensionale Bisektionsverfahren gespeichert.

Nun zur naiven Implementierung des Bisektionsverfahrens. Diese weicht von Listing 10 nur sofern ab, dass uns auch Bisektion mehrerer Tetraeder erlaubt ist, wobei dies durch mehrfaches Ausführen der Listing 10 in einer geeigneten Schleife zustandekommt. Außerdem werden Randflächen mitverfeinert.

Listing 12: REFINE

```
function [elements,elementgeneration,coordinates,varargout] ...
                 = refine(elements,elementgeneration,coordinates,varargin)
nB = nargin-4;
[varargout{1:nB}] = varargin{1:nB};%* varargout{1:nB}==boundaries{1:nB}
markedElements = unique(varargin{end});
%*** Wrapper-Schleife
while ~isempty(markedElements)
  marked = markedElements(1);
  nE = size(elements,1);
  element2neighbors = provideNeighbors(elements);
  refined = [];
  K = [];
  F = marked;
  while (~isempty(F))
    Fnew = [];
    for T1 = F
       for T2 = setdiff(element2neighbors(T1,[2,3]),[F,K,0])%** T2 ∈ N(P,T1)\(F∪K)
        if (elementgeneration(T2)==elementgeneration(T1))%** kompatibel teilbar
          Fnew = union(Fnew,T2);
        else
          [elements,elementgeneration,coordinates,varargout{1:nB}...
                ,children,refinedElements] = refine(elements...
                ,elementgeneration,coordinates,varargout{1:nB},T2);
          refined = [refined,refinedElements];
          element2neighbors = provideNeighbors(elements);
          nE = size(elements,1);
          if length(intersect(elements(T1,:),elements(child(1),:)))==3
            Fnew = union(Fnew,children(1));%** linkes Kind
          else
            Fnew = union(Fnew,children(2));%** rechtes Kind
          end
        end
       end
    end
    K = union(K,F);
    F = Fnew;
  end
  %*** Erzeugung des neuen Knotens
  coordinates(end+1,:) = 1/2*( coordinates(elements(K(1),1),:)...
                             + coordinates(elements(K(1),4),:) );
  newNode = size(coordinates,1);
```

```
42    %*** Verfeinerung der zur Kante gehoerigen Elemente
43    K = [marked,setdiff(K,marked)];%*** Damit children==[marked,nE+1] gilt
44    if (mod(elementgeneration(K),3)==0)
45      elements([K,(nE+(1:length(K)))],:) ...
46        = [ elements(K,1),newNode*ones(length(K),1),elements(K,[2,3]);...
47            elements(K,4),newNode*ones(length(K),1),elements(K,[3,2]) ];
48    else
49      elements([K,(nE+(1:length(K)))],:) ...
50        = [ elements(K,1),newNode*ones(length(K),1),elements(K,[2,3]);...
51            elements(K,4),newNode*ones(length(K),1),elements(K,[2,3]) ];
52    end
53    elementgeneration([K,nE+(1:length(K))]) ...
54         = [ elementgeneration(K)+1,elementgeneration(K)+1 ];
55   %*** Verfeinerung der zur Kante gehoerigen Flaechen
56    for j = 1:min(nargout-3,nB)
57      if ~isempty(varargout{j})
58        nBE = size(varargout{j},1);
59        markedFaceNode = ( (varargout{j}==elements(K(1),1)) | ...
60                           (varargout{j}==elements(K(1),4)) );
61        bisec = find(markedFaceNode(:,1) & markedFaceNode(:,3));
62        if ~isempty(bisec)
63          varargout{j}([bisec,nBE+(1:length(bisec))'],:) =  ...
64                  [ varargout{j}(bisec,1),newNode*ones(length(bisec),1),...
65                    varargout{j}(bisec,2);...
66                    varargout{j}(bisec,3),newNode*ones(length(bisec),1),...
67                    varargout{j}(bisec,2) ];
68        end
69      end
70    end
71    %*** Hintergrundinformationen fuer die Rekursion
72    children = [marked,nE+1];
73    varargout{nB+1} = children;
74    refined = [refined,K];
75    varargout{nB+2} = refined;
76    markedElements = setdiff(markedElements,refined);
77  end
```

- Zeile 1: Die Funktion `refine` stellt eine MATLAB-Implementierung der Listing 10 dar. Sie übernimmt eine reguläre Triangulierung, die den Bedingungen aus Satz 5.3.7 genügt (das sind insbesondere alle durch `validStartMesh` erzeugte und von `refine` weiterbearbeitete Triangulierungen), sowie ein Array `marked`, das die Nummern der Elemente enthält, die verfeinert werden sollen. Die Funktion wird üblicherweise so aufgerufen:

  ```
  [elements,elementgeneration,coordinates,neumann,dirichlet] = ...
  refineC(elements,elementgeneration,coordinates,neumann,dirichlet,marked)
  ```

- Zeilen 8–37: Hier werden gänzlich analog zur Listing 10 alle Elemente gesucht, die zur aktuellen Referenzkante gehören. Es muss dabei darauf geachtet werden, dass das korrekte Kind (Zeile 22, 27–31) gewählt wird. Außerdem müssen die Elemente `refinedElements` gespeichert werden, die vom Rekursionsschritt verfeinert wurden um sie später aus der Liste der noch zu verfeinernden Tetraeder zu streichen.

- Zeilen 38–41: Der Halbierungspunkt der zu den Elementen aus `K` gehörigen Verfeinerungskante wird generiert.

- Zeilen 42–54: Hier werden gemäß dem Verfeinerungsschema aus Definition 5.3.2 gleichzeitig alle Elemente aus K geteilt. Die linken Kinder werden an die Stelle der Elterntetraeder

geschrieben und die rechten an `elements` angehängt. Dadurch bleibt unsere Konvention aus Bemerkung 5.3.9 erhalten.

- Zeilen 56–70: Hier werden die Randflächen unseren Konventionen konform verfeinert.
- Zeilen 72–76: Hier werden die Informationen `children` und `refinedElements` aufgebaut, die für die Rekursion benötigt werden. Weiters werden die soeben verfeinerten Elemente aus dem Array `markedElements` entfernt, damit die `for`-Schleife aus Zeile 7 terminiert sobald es leer ist.

Abschließend sei betont, dass diese Implementierung keineswegs effizient ist. Dafür gibt es mehrere Gründe.

- Die Bestimmung eines Nachbarn findet nicht in konstanter Zeit statt. Die Datenstruktur `element2neighbors` wird für jede Kante, die im Zuge des Verfahrens geteilt werden muss neu generiert. Mit e der Anzahl der Kanten, die in Summe geteilt werden und n der Anzahl der Tetraeder in der ursprünglichen Triangulierung, kann somit die Laufzeit grob durch $\mathcal{O}(en \log n)$ nach unten abgeschätzt werden.
- Die Datenstrukturen `elements`, `elementtype`, `coordinates`, `dirichlet` und `neumann` werden e-mal neu erstellt.
- Nachdem in Zeile 38 die Elemente K bestimmt wurden, die verfeinert werden sollen, muss in Zeilen 59–61 trotzdem das gesamte Array `varargout` durchsucht werden um die zugehörigen Flächen zu bestimmen.

Das Verfahren eignet sich vor allem für Implementierungen mit einer Struktur in der eine Vielzahl von Änderungen effizient geschehen kann und speziell die Nachbarschaftsstrukturen bei jeder Verfeinerung mitaktualisiert werden. Insbesondere ein Binärbaum, dessen Blätter die einzelnen Tetraeder darstellen, die wiederum untereinander durch Nachbarschaftsrelationen verknüpft sind, eignet sich besonders dafür. So eine Implementierung ist auch im ursprünglichen Artikel [15] zu finden. Ob der Algorithmus auf ähnliche Weise wie in [6] vektorisiert werden kann, ist im Zuge dieser Arbeit ungeklärt geblieben. Das zweidimensionale Verfahren aus [6] basiert dabei darauf zuerst alle Kanten zu markieren, die im Zuge der Verfeinerung geteilt werden müssen und anschließend für jedes Element abhängig von den markierten Kanten eines von $2^{\binom{d+1}{2}-1} = 4$ Verfeinerungsschemen anzuwenden. Für das Verfahren aus Listing 10 ist allerdings nicht klar, ob es grundsätzlich hinreichend ist nur Kanten der Starttriangulierung zu markieren, oder ob auch gegebenenfalls durch den rekursiven Prozess entstandene Kanten markiert werden müssten. Im Falle, dass eine Kantenmarkierung hinreichend ist müsste zwischen $2^{\binom{d+1}{2}-1} = 32$ möglichen Verfeinerungsschemen für jeden der drei Tetraedertypen unterschieden werden. Falls sie allerdings nicht hinreichend ist, kann das Verfahren nicht auf diese Weise vektorisiert werden. Für eine effiziente Implementierung von Listing 10 in MATLAB kann jedoch die `mex`-Schnittstelle verwendet werden, die uns erlaubt C-Code einzubinden. Der entsprechende Programmcode ist im Anhang zu finden.

5.4 Vergröbern von Triangulierungen

Nun kommen wir zur bereits mehrmals erwähnten Vergröberung. Diese ist vor allem bei zeitabhängigen Problemen von Interesse, bei denen sich der Bereich ändert, an dem ein hoher Verfeinerungsgrad benötigt wird. Da wir nach einem Vergröberungsschritt gegebenenfalls wieder weiterverfeinern wollen müssen wir darauf achten, dass unsere Vergröberungsmethode auf das Bisektionsverfahren abgestimmt ist. Als Vergröberung lassen wir also nur das Zusammenführen

zweier Kinder zu, die im Zuge eines früheren Bisektionsschrittes aus demselben Elterntetraeder gewonnen wurden. Wenn dies nun auch so geschieht, dass dabei die Triangulierung regulär bleibt, dann erhalten wir wieder eine Triangulierung, die wir für `refine` weiterverwenden können. Dies erklären wir als unser Ziel. Die Anforderungen, die wir an das Vergröberungsverfahren stellen, unterscheiden sich jedoch etwas von denen an die Funktion `refine`. Wenn wir dort einen Tetraeder T zur Verfeinerung markieren, dann sollen von der Funktion `refine` alle Tetraeder verfeinert werden, deren Bisektion für die Teilung von T nötig ist. Die Funktion `coarsen` jedoch soll keine Elemente vergröbern, die nicht zur Vergröberung markiert sind. Ein Knoten soll insbesondere nur dann vergröbert werden, wenn auch alle anliegenden Tetraeder dafür markiert wurden. Das darf auch durchaus dazu führen, dass nicht alle gewünschten Tetraeder von der Funktion `coarsen` vergröbert werden. Wann nun aber ein Knoten durch Vergröberung der anliegenden Tetraeder aus der Triangulierung entfernt werden darf, zeigt der folgende Satz. Die darin gebrachte Bedingung ist klarerweise eine notwendige Bedingung für das Vergröbern, sie ist aber sogar hinreichend.

Satz 5.4.1 (Bartels & Schreier, [2]). *Ein Knoten einer durch Bisektion erhaltenen Triangulierung, der nicht zur Starttriangulierung gehört, kann genau dann vergröbert werden, wenn er der neueste Knoten aller Elemente ist zu denen er gehört.*

Diesen Satz gießen wir nun in Codeform. Der Funktion `coarsen` soll neben einer durch Bisektion entstandenen Triangulierung im üblichen Format eine Liste von zu vergröbernden Tetraedern `marked` übergeben werden können. Die Funktion soll sodann alle Elemente dieser Liste vergröbern, deren neueste Knoten die Bedingung aus Satz 5.4.1 erfüllen. Auf eine anschließende Überprüfung, ob danach eventuell weitere Tetraeder aus `marked` diese Bedingung erfüllen, verzichten wir. Die Vergröberung ist somit nicht im vollen Ausmaß invers zur Verfeinerung, denn im allgemeinen sind mehrere Vergröberungsschritte notwendig um einen Aufruf der Funktion `refine` rückgängig zu machen.

Listing 13: Coarsen

```
function [elements,elementgeneration,coordinates,varargout] = ...
                    coarsen(elements,elementgeneration,coordinates,varargin)
nC = size(coordinates,1);
nE = size(elements,1);
nB = nargin-4;
marked = varargin{end};
%*** Bestimmung vergroeberbarer Knoten
node2numTet = accumarray(reshape(elements,[],1),1,[nC 1]);
markednewestV2numTet = accumarray(elements(marked,2),1,[nC 1]);
possibleNodes = (node2numTet == markednewestV2numTet);
%*** Bestimmung vergroeberbarer Elemente
element2neighbors = provideNeighbors(elements);
leftHalf = ( (elementgeneration > 2) & possibleNodes(elements(:,2))...
           & (element2neighbors(:,1) > (1:nE)') );
%*** Festlegen der zu entfernenden Knoten
markedNodes = zeros(nC,1);
markedNodes(elements(leftHalf,2)) = 1;
%*** Erzeugung der neuen Koordinatennummern
coordinates = coordinates(~markedNodes,:);
coordinates2newCoordinates = zeros(1,nC);
coordinates2newCoordinates(~markedNodes) = 1:size(coordinates,1);
%*** Vergroeberung der Triangulierung
rightHalf = element2neighbors(leftHalf,1);
elements(leftHalf,:) = [ elements(leftHalf,[1,3,4]),...
                         elements(rightHalf,1) ];
```

```
26 elements(rightHalf,:) = [];
27 elementgeneration(leftHalf) = elementgeneration(leftHalf)-1;
28 elementgeneration(rightHalf) = [];
29 elements = coordinates2newCoordinates(elements);
30 %*** Vergroeberung der Randflaechen
31 for j = 1:min(nargout-3,nB)
32     if ~isempty(varargin{j})
33         nBE = size(varargin{j},1);
34         varargout{j} = varargin{j};
35         boundary2neighbors = provideNeighbors(varargin{j});
36         leftHalf = ( markedNodes(varargin{j}(:,2)) ...
37                    & (boundary2neighbors(:,1) > (1:nBE)') );
38         rightHalf = boundary2neighbors(leftHalf,1);
39         varargout{j}(leftHalf,:) = ...
40                 [varargout{j}(leftHalf,[1,3]),varargout{j}(rightHalf,1)];
41         varargout{j}(rightHalf,:) = [];
42         varargout{j} = coordinates2newCoordinates(varargout{j});
43     end
44 end
```

- Zeilen 1–2: Der Funktion `coarsen` wird eine Triangulierung und die Liste der zu vergröbernden Elemente als Array `marked=varargin{end}` von Indizes in `elements` übergeben. Sie führt einen Vergröberungsschritt durch und liefert das entstandene Netz zurück. Falls gewisse Elemente nicht mit einem Schritt vergröbert werden können, bleiben diese unverändert.

- Zeilen 8–11: Hier werden mit Satz 5.4.1 die neuesten Knoten der Tetraeder aus `marked` überprüft. Dazu werden in Zeile 8 die Anzahlen der an den Knoten adjazenten Tetraeder generiert. In Zeile 9 werden zu allen neuesten Knoten die Anzahlen der adjazenten Tetraeder aus `marked` bestimmt. Der Vergleich dieser beiden Anzahlen liefert in Zeile 12 dann die Knoten, die entfernt werden können, falls sie nicht zur Starttriangulierung gehören.

- Zeilen 12–17: Nun wird mittels `elementgeneration` überprüft ob die Elemente zur Starttriangulierung gehören. (Anm.: Hier wird implizit angenommen, dass zur Generierung einer passenden Starttriangulierung die Funktion `validStartMesh` verwendet wurde. Dies führt zur Vereinfachung des Codes, es können aber somit anderwertig generierte Starttriangulierungen nicht bis zur Generation 0 vergröbert werden. In den Augen des Autors ist dies allerdings ein vernachlässigbarer Nachteil, da es sich einerseits nur um zwei Vergröberungsschritte handelt und andererseits die Suche nach einer Starttriangulierung ohne Zuhilfenahme der Funktion `validStartMesh` ohnehin praktisch nicht relevant ist.) Falls die Elemente nicht zur Starttriangulierung gehören wird der erste der beiden Geschwistertetraeder in das Array `leftHalf` aufgenommen. Beachte dabei, dass hier unsere Konvention wichtig ist, dass das linke Kind eines Tetraeders immer vor dem rechten Kind in `elements` auftritt, damit die Vergröberungsregel den korrekten Elterntetraeder liefert. In Zeile 17 werden die zu löschenden Knoten entsprechend angepasst.

- Zeilen 19–21: Die nicht mehr gebrauchten Knoten werden gelöscht und es wird eine neue Nummerierung der verbleibenden Knoten erstellt.

- Zeilen 23–29: Die Geschwister werden zu den Elternelementen zusammengeführt. Indem dabei die Eltern an die Positionen der linken Kinder geschrieben werden, bleibt die Eigenschaft erhalten, dass linke Kinder immer vor rechten Kindern stehen. Die alten Elemente werden gelöscht und `elements` an die neuen Knotennummerierungen angepasst.

- 31–44: Durch die von `elements` unabhängige Speicherung im zweidimensionalen Bisektionsformat kann die Vergröberung der Randflächen gänzlich analog zu den Zeilen 12–29 erfolgen.

Wir erhalten also nach Anwendung von `coarsen` eine reguläre Triangulierung, die wieder unseren Konventionen genügt.

6 A posteriori Fehlerschätzer und adaptives Verfahren

In diesem Abschnitt werden wir die kennengelernten Funktionen zu einem gekapselten Verfahren abrunden. Dazu benötigen wir Methoden um numerisch berechenbare Schranken $\eta = \eta(U, f, g, \mathcal{T})$ für den Fehler $||u - U||_{H^1(\Omega)}$ zu finden, die zwar von der diskreten Lösung U, der Triangulierung $\mathcal{T}$ und den Daten f und g abhängen, aber *nicht* von der exakten Lösung u. Wir nennen so eine Größe η einen *(a posteriori) Fehlerschätzer*. Genauer wollen wir zwei Eigenschaften eines Fehlerschätzers definieren:

Definition 6.0.2. *Ein Fehlerschätzer $\eta_\ell(U_\ell, f, g, \mathcal{T})$ heißt* zuverlässig*, falls es ein C_{zuv} gibt, sodass $C_{zuv}\,\eta_\ell$ eine obere Schranke für den Fehler $||u - U_\ell||_{H^1(\Omega)}$ ist. Falls es ein C_{eff} gibt, sodass $C_{eff}\,\eta_\ell$ eine untere Schranke für den Fehler ist, dann nennen wir η_ℓ* effizient*.*

Für einen zuverlässigen Fehlerschätzer η_ℓ folgt also aus der Konvergenz von η_ℓ gegen Null die Konvergenz von U gegen die Lösung u. Falls η_ℓ auch noch effizient ist, konvergiert der Fehlerschätzer sogar mit derselben Ordnung wie $||u-U||_{H^1(\Omega)}$. Bei der Berechnung von Fehlerschätzern stecken wir uns nun zwei Ziele:

- Wir wollen die Genauigkeit $||u-U||_{H^1(\Omega)}$ der diskreten Lösung U steuern und die Berechnung beenden, sobald wir die gewünschte Genauigkeit erreicht haben.
- Die Netzverfeinerung sollte automatisch vom Algorithmus geregelt werden, um den größtmöglichen Exaktheitsgrad abhängig von der Anzahl der Elemente zu erreichen.

In Anbetracht des zweiten Zieles ist es also insbesondere wichtig auch Aussagen über den Fehler auf einzelnen Elementen der Triangulierung treffen zu können. Größen η_T, für die heuristisch gilt

$$\eta_T \approx ||u - U||_{H^1(T)} \text{ für alle } T \in \mathcal{T}, \tag{6.0.1}$$

bezeichnen wir als *Verfeinerungsindikatoren*. Der zugehörige Fehlerschätzer ist dann durch $\eta := \left(\sum_{T\in\mathcal{T}} \eta_T^2\right)^{1/2}$ definiert. Wir werden nun zuerst das allgemeine adaptive Verfahren besprechen, das unabhängig von der Wahl des Fehlerschätzers gleich aufgebaut ist. Dann werden wir als konkretes Beispiel einen residualen Fehlerschätzer und dessen Implementierung behandeln.

6.1 Ein adaptives Verfahren

Zu gegebenen Verfeinerungsindikatoren $\eta_T \approx ||u - U||_{H^1(T)}$, markieren wir eine Menge $\mathcal{M} \subseteq \mathcal{T}$ von Tetraedern zur Verfeinerung. Wir wählen dabei $\mathcal{M}$ so, dass es eine Menge kleinster Kardinalität ist, für die

$$\rho \sum_{T\in\mathcal{T}} \eta_T^2 \leq \sum_{T\in\mathcal{M}} \eta_T^2 \tag{6.1.1}$$

mit einem gewissen Parameter $\rho \in (0, 1)$ gilt. Dieses Kriterium ist als Dörfler-Kriterium bekannt und liefert bei sukzessiver Anwendung Konvergenz mit optimaler Rate, falls das Problem

$\Gamma = \Gamma_D$ und $u_D = 0$ erfüllt (siehe dazu [5]). Nach der Bestimmung von $\mathcal{M}$ wird eine neue Triangulierung $\mathcal{T}'$ aus $\mathcal{T}$ generiert, indem (zumindest) die markierten Tetraeder $T \in \mathcal{M}$ geteilt werden. Beachte dabei, dass für $\rho \to 1$ nahezu uniforme Verfeinerungen entstehen, wohingegen die Triangulierungen für $\rho \to 0$ hoch adaptiv werden.

Listing 14: Adaptives Verfahren

```
1  function [x,coordinates,elements,indicators] = adaptiveAlgorithm(elements,...
2              elementgeneration,coordinates,dirichlet,neumann,f,g,uD,nEmax,rho)
3  while 1
4      %*** Berechnung der diskreten Loesung
5      x = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD);
6      %*** Berechnung der Verfeinerungsindikatoren
7      indicators = computeEtaR(x,coordinates,elements,dirichlet,neumann,f,g);
8      %*** Abbruchkriterium
9      if size(elements,1) >= nEmax
10         break
11     end
12     %*** Elemente zur Verfeinerung markieren
13     [indicators,idx] = sort(indicators,'descend');
14     sumeta = cumsum(indicators);
15     ell = find(sumeta>=sumeta(end)*rho,1);
16     marked = idx(1:ell);
17     %*** Triangulierung verfeinern
18     [elements,elementgeneration,coordinates,dirichlet,neumann] = ...
19        refine(elements,elementgeneration,coordinates,dirichlet,neumann,marked);
20 end
```

- Zeilen 1–2: Die Funktion übernimmt eine für Bisektion passende Starttriangulierung im üblichen Format `elements`, `elementgeneration`, `coordinates`, `dirichlet` und `neumann`, sowie die Problemdaten f,g und u_D. Außerdem übergibt der Benutzer die maximale Anzahl der Elemente `nEmax` sowie den Parameter ρ aus Gleichung (6.1.1). Zurückgegeben wird der Koeffizientenvektor $\mathbf{x}$ der letzten diskreten Lösung $U \in S^1_D(\mathcal{T})$, die zugehörige letzte Triangulierung $\mathcal{T}$ und der Vektor der elementweisen Fehlerindikatoren `indicators`.
- Zeilen 3–20: Solange die Anzahl der Elemente der Triangulierung $\#\mathcal{T}$ kleiner ist als die vorgegebene Schranke `nEmax` gehen wir wie folgt vor: Wir berechnen die diskrete Lösung (Zeile 5) und den Vektor der Verfeinerungsindikatoren (Zeile 7), dessen j-ter Eintrag den Wert $\eta^2_{T_j}$ speichert. In den Zeilen 13–16 wird das Dörfler-Kriterium realisiert. Durch absteigende Sortierung der Verfeinerungsindikatoren (Zeile 13) lässt sich in den beiden darauffolgenden Zeilen leicht die gesuchte Menge $\mathcal{M}$ bestimmen. In den Zeilen 18–19 generieren wir durch Verfeinerung der markierten Elemente eine neue Triangulierung.

6.2 Residuum-basierter Fehlerschätzer

Wir betrachten nun den Fehlerschätzer $\eta_R := \left(\sum_{T\in\mathcal{T}} \eta_T^2\right)^{1/2}$ mit den Verfeinerungsindikatoren

$$\eta_T^2 := h_T^2 \left\|f\right\|^2_{L^2(T)} + h_T \left\|J_h(\partial_n U)\right\|^2_{L^2(\partial T\cap\Omega)} + h_T \left\|g - \partial_n U\right\|^2_{L^2(\partial T\cap\Gamma_N)}. \tag{6.2.1}$$

Dabei bezeichne $J_h(.)$ den Sprung über eine innere Fläche $F \in \mathcal{F}$, $F \notin \Gamma$ und h_T die Länge der längsten Kante von T. Für benachbarte Elemente $T_\pm \in \mathcal{T}$ mit äußeren Normalvektoren $n_\pm$ ist der Sprung der $\mathcal{T}$-stückweise konstanten Funktion ∇U über die gemeinsame Fläche $F = T_+ \cap T_- \in \mathcal{F}$ definiert durch

$$J_h(\partial_n U)|_F := \nabla U|_{T_+} \cdot n_+ + \nabla U|_{T_-} \cdot n_-, \tag{6.2.2}$$

was wegen $n_+ = -n_-$ eigentlich eine Differenz darstellt. Wir werden den ersten Summanden aus (6.2.1) als Volumenbeitrag und die beiden anderen Summanden als Flächenbeitrag bezeichnen. Von obigem Fehlerschätzer kann gezeigt werden, dass er zuverlässig und effizient ist (Siehe [16, Abschnitt 2]), genauer, dass

$$C_{\text{rel}}^{-1}||u-U||_{H^1(\Omega)} \leq \eta_R \leq C_{\text{eff}}\big(||u-U||_{H^1(\Omega)} + ||h(f-f_{\mathcal{T}})||_{L^2(\Omega)} + ||h^{1/2}(g-g_{\mathcal{F}})||_{L^2(\Gamma_N)}\big) \tag{6.2.3}$$

gilt, wobei die Konstanten $C_{\text{rel}}, C_{\text{eff}} > 0$ nur von der Form der Elemente aus $\mathcal{T}$, dem Gebiet Ω und den Daten f und g abhängen. Dabei sind $f_{\mathcal{T}}$ und $g_{\mathcal{F}}$ die $\mathcal{T}$-elementweisen bzw. $\mathcal{F}$-flächenweisen Integralmittel von f und g. Bemerke, dass für glatte Daten gilt $||h(f-f_{\mathcal{T}})||_{L^2(\Omega)} = \mathcal{O}(h^2)$ und $||h^{1/2}(g-g_{\mathcal{F}})||_{L^2(\Gamma_N)} = \mathcal{O}(h^{3/2})$, sodass diese Terme von höherer Ordnung als der Fehler $||u-U||_{H^1(\Omega)}$ sind.

Implementiert wird nun folgende Annäherung:

$$\tilde{\eta}_T^2 := |T|^{2/3}Q_{n,T}^{\triangle}(f^2) + \sum_{F\in\partial T\cap\Omega}\frac{|T|}{|F|}Q_{n,F}^{\triangle}((J_h(\partial_n U)|_F)^2) + \sum_{F\in\partial T\cap\Gamma_N}\frac{|T|}{|F|}Q_{n,F}^{\triangle}((g-\partial_n U|_F)^2). \tag{6.2.4}$$

Formregularität der Triangulierung garantiert dabei, dass

$$\frac{|T|}{|F|} \leq h_T \leq C\frac{|T|}{|F|} \text{ und } |T|^{2/3} \leq h_T^2 \leq C|T|^{2/3} \text{ für alle } T\in\mathcal{T} \text{ mit Flächen } F\subset\partial T \tag{6.2.5}$$

für eine gewisse Konstante $C > 0$ gilt und dass somit die Verfeinerungsindikatoren $\tilde{\eta}_T$ und η_T äquivalent sind. Nun zur Implementierung:

Listing 15: Residualschätzer nicht vektorisiert

```
function etaR = computeEtaR(x,elements,coordinates,dirichlet,neumann,f,g)
quaddeg = 3;
[face2nodes,element2faces,dirichlet2face,neumann2face] = ...
                                        provideFaceData(elements,dirichlet,neumann);
nE = size(elements,1);
nF = size(face2nodes,1);
etaR = zeros(nE,1);
jumps = zeros(nF,1);
%*** Berechnung der Normalenableitungen fuer alle Flaechen
for j = 1:nE
    nodes = elements(j,1:4);
    B = [1 1 1 1; coordinates(nodes,:)'];
    G = B \ [0 0 0; 1 0 0; 0 1 0; 0 0 1];
    grad = G'*x(nodes);
    for k = 1:4
        face = element2faces(j,k);
        %*** Berechnung der ausseren Normalen
        outdirection = coordinates(face2nodes(face,1),:)-coordinates(nodes(k),:)
        normal = cross([1,0,-1]*coordinates(face2nodes(face,:),:),...
                       [0,1,-1]*coordinates(face2nodes(face,:),:) );
        normal = normal/norm(normal)*sign(dot(normal,outdirection));
        jumps(face) = jumps(face) + normal*grad;
    end
end
faceEtaR = jumps.^2;
%*** Loeschen der Daten im Falle von Dirichlet-Randflaechen
for j = 1:size(dirichlet,1)
```

```
28      faceEtaR(dirichlet2face(j)) = 0;
29  end
30  %*** Beitraege auf dem Neumann-Rand
31  for j = 1:size(neumann,1)
32      face = neumann2face(j);
33      [XYZ,W,sizeF] = triquad(quaddeg,coordinates(face2nodes(face,:),:));
34      faceEtaR(face) = (1/sizeF)*W*((repmat(jumps(face),quaddeg^2,1)-g(XYZ)).^2);
35  end
36  %*** Bestimmung des Residualschaetzers
37  for j = 1:nE
38      nodes = elements(j,:);
39      [XYZ,W,sizeT] = tetquad(quaddeg,coordinates(nodes,:));
40      fintegral = W*(feval(f,XYZ).^2);
41      etaR(j) = sizeT^(2/3)*fintegral;
42      for k = 1:4
43          face = element2faces(j,k);
44          etaR(j) = etaR(j) + sizeT*faceEtaR(face);
45      end
46  end
```

- Zeile 1: Die Funktion `computeEtaR` übernimmt den Koeffizientenvektor `x` der diskreten Lösung, die Triangulierung in üblicher Form als `elements`, `coordinates`, `dirichlet`, `neumann` und die Funktionen `f` und `g`. Zurückgegeben wird ein Spaltenvektor der elementweisen Fehlerschätzer.

- Zeilen 10–25: Hier werden die Sprungterme aus (6.2.2) berechnet. Für jedes Element werden in Zeile 13 die Gradienten der vier zugehörigen Hutfunktionen in `G` erzeugt. Das 3×1-Array `grad` beinhaltet dann den Gradienten der diskreten Lösung auf dem jeweiligen Tetraeder. In den Zeilen 15–22 werden dann die äußeren Normalen der Tetraederrandflächen gebildet und die Sprungterme in `jumps` gespeichert. Beachte, dass für eine Flächen `face` die auf dem Rand des Gebietes liegt der Eintrag `jumps(face)` eigentlich nur die Normalenableitung beinhaltet. Dieses Faktum wird in den Zeilen 31–35 verwendet. Zuvor wird allerdings noch der Array `faceEtaR` aufgebaut, der in Zeile 36 die notwendigen Ausdrücke zur Berechnung des Flächenbeitrags enthalten wird. Die überflüssig berechneten Terme für den Dirichlet-Rand werden in Zeile 28 gelöscht. Die anderen Terme werden in Zeile 25 quadriert und in `faceEtaR` übernommen.

- Zeilen 31–35: Hier wird der Summand des Flächenbeitrags berechnet, der am Neumann-Rand lebt. Dazu wird eine Gauß-Quadratur der Funktion $(g - \partial_n U)^2$ durchgeführt. Wobei die flächenweise konstante Normalenableitung $\partial_n U$ als `jumps(face)` vorhanden ist. Zusätzlich wird durch den Flächeninhalt der jeweilgen Randfläche dividiert. Dies ist wichtig, denn die weiter oben berechneten Einträge aus faceEtaR wurden weder mit dem Flächeninhalt noch h_T skaliert. In Zeile 36 enhält `faceEtaR` für eine Fläche F also die Einträge

$$(J_h(\partial_n U)|_F)^2 = \frac{1}{|F|} Q^{\triangle}_{n,F}((J_h(\partial_n U)|_F)^2) \quad \text{bzw.} \quad \frac{1}{|F|} Q^{\triangle}_{n,F}((g - \partial_n U|_F)^2). \tag{6.2.6}$$

- Zeilen 37–46: Hier werden nun für jeden Tetraeder T die zugehörigen Verfeinerungsindikatoren berechnet. Dazu muss nur mehr eine Quadratur der Funktion f^2 realisiert werden und die mit $|T|$ skalierten Flächenbeiträge dazusummiert werden. Schließlich erhalten wir die in (6.2.4) genannten Fehlerindikatoren.

Auch die Funktion `computeEtaR` kann mittels Vektorisierung erheblich beschleunigt werden. Wir besprechen nun Listing 16.

Listing 16: Residualschätzer vektorisiert

```
function etaR = computeEtaR(x,elements,coordinates,dirichlet,neumann,f,g)
quaddeg = 3;
[face2nodes,element2faces,dirichlet2face,neumann2face] = ...
                                        provideFaceData(elements,dirichlet,neumann);
nE = size(elements,1);
nF = size(face2nodes,1);
nN = size(neumann,1);
nD = size(dirichlet,1);
%*** Berechnung des Gradienten
C1 = coordinates(elements(:,1),:);
C2 = coordinates(elements(:,2),:);
C3 = coordinates(elements(:,3),:);
C4 = coordinates(elements(:,4),:);
  %* Normalvektoren
N1 = -cross(C3-C2,C4-C2,2);
N2 =  cross(C4-C3,C1-C3,2);
N3 = -cross(C1-C4,C2-C4,2);
N4 =  cross(C2-C1,C3-C1,2);
  %* Gradienten der Hutfunktionen
G1 = N1./repmat(dot(N1,(C1-C4),2),1,3);
G2 = N2./repmat(dot(N2,(C2-C1),2),1,3);
G3 = N3./repmat(dot(N3,(C3-C2),2),1,3);
G4 = N4./repmat(dot(N4,(C4-C3),2),1,3);
  %* Gradient der durch x repraesentierten diskreten Loesung U
X1 = x(elements(:,1));
X2 = x(elements(:,2));
X3 = x(elements(:,3));
X4 = x(elements(:,4));
G = repmat(X1,1,3).*G1 + repmat(X2,1,3).*G2 ...
  + repmat(X3,1,3).*G3 + repmat(X4,1,3).*G4;
volumes = (1/6)*abs(dot(N1,C1-C2,2));
%*** Berechnung der Sprungterme
  %* Beachte: -Gi./sqrt(sum(Gi.^2,2)) ist normierte aussere Normale
dudn1 = -dot(G1,G,2)./sqrt(sum(G1.^2,2));
dudn2 = -dot(G2,G,2)./sqrt(sum(G2.^2,2));
dudn3 = -dot(G3,G,2)./sqrt(sum(G3.^2,2));
dudn4 = -dot(G4,G,2)./sqrt(sum(G4.^2,2));
jumps = accumarray(element2faces(:),[dudn1;dudn2;dudn3;dudn4],[nF,1]);
faceEtaR = jumps.^2;
%*** Loeschen der Daten im Falle von Dirichlet-Randflaechen
if nD~=0
    faceEtaR(dirichlet2face) = 0;
end
%*** Beitraege auf dem Neumann-Rand
if nN~=0
    TriMesh = TriRep(neumann,coordinates);
    [~,W,~,V] = triquad(quaddeg,[]);%* Liefert Gewichte fuer |F| = sum(W) = 1/2
    Vrep = repmat(V,nN,1);
    XYZ = baryToCart(TriMesh, reshape(repmat(1:nN,quaddeg.^2,1),[],1), Vrep);
    %* Integrale sind schon durch Flaecheninhalt dividiert
    faceEtaR(neumann2face(:)) = 2*W*reshape(...
        (reshape(repmat(jumps(neumann2face)',quaddeg^2,1),[],1)-g(XYZ)).^2,[],nN);
end
%*** Volumenbeitraege bestimmen
TetMesh = TriRep(elements,coordinates);
[~,W,~,V] = tetquad(quaddeg,[]);% Liefert Gewichte fuer |T| = sum(W) = 1/6
Vrep = repmat(V,nE,1);
XYZ = baryToCart(TetMesh, reshape(repmat(1:nE,quaddeg.^3,1),[],1), Vrep);
fintegrals = volumes.*(6*W*reshape(feval(f,XYZ).^2,[],nE))';
```

```
60 %*** Bestimmung des Residualschaetzers
61 etaR = volumes.^(2/3).*fintegrals + ...
62         volumes.*sum(reshape(faceEtaR(element2faces),nE,[]),2);
```

- Zeile 1: Die Funktion `computeEtaR` übernimmt den Koeffizientenvektor `x` der diskreten Lösung, die Triangulierung in üblicher Form als `elements`, `coordinates`, `dirichlet`, `neumann` und die Funktionen `f` und `g`. Zurückgegeben wird ein Spaltenvektor der elementweisen Fehlerschätzer.
- Zeilen 10–31: Hier wird der Gradient der diskreten Lösung berechnet. Der (`nE` $\times 3$)-Array `G` aus Zeile 29 enthält die elementweise konstanten Gradienten. Zu deren Berechnung werden zuerst die Arrays der Normalvektoren `N1`, `N2`, `N3`, `N4` und darauf aufbauend in Zeilen 20–23 die Gradienten der Hutfunktionen berechnet. Der Gradient der diskreten Lösung ergibt sich dann als Linearkombination ebendieser.
- Zeilen 34–39: Hier werden die Sprungterme berechnet. Anstatt aber zuerst die Normalvektoren zu normieren und nach außen zu orientieren kann man hier anders vorgehen. Die Gradienten der Hutfunktionen sind nämlich nach innen orientierte Normalvektoren. Diese werden also nach außen orientiert und normiert. Dies kann wegen der Bilinearität des Skalarproduktes auch nach Aufruf von `dot` geschehen. Es ergeben sich also die Zeilen 34–37. Der Array `jumps` wird mit dem Befehl `accumarray` erzeugt.
- Zeilen 45–53: Diese Zeilen entsprechen den Zeilen 31–35 aus der vorangehenden Listing. Mittels Gauß-Quadratur wird die Funktion $(g - \partial_n U)^2$ integriert.
- Zeilen 55–62: Hier werden die Volumenbeiträge durch Quadratur von f^2 bestimmt und schlussendlich die Verfeinerungsindikatoren zusammengesetzt.

6.3 Numerische Experimente

Abschließend wollen wir die Vorteile des adaptiven Verfahrens anhand eines Beispiels verdeutlichen. Die folgenden Berechnungen wurden dabei mittels MATLAB Version 7.12 (R2011a) auf einem Rechner mit Intel Core(tm) 2 Duo Prozessor (2.16 GHz) und 3 GB RAM durchgeführt. Als Gebiet wählen wir $\Omega = (-1,1)^3 \backslash [0,1]^3$, den sogenannten Fichera-Würfel. Die Unterteilung des Randes $\Gamma = \partial\Omega$ in Neumann- und Dirichlet-Rand ist der Abbildung 16 zu entnehmen. Wir lösen Problem (1.0.1) mit rechter Seite

$$f(x,y,z) := -\frac{5}{16}\,(x^2+y^2+z^2)^{-7/8}. \tag{6.3.1}$$

Die Randdaten u_D und g sind entsprechend der exakten Lösung

$$u(x,y,z) = (x^2+y^2+z^2)^{1/8} \tag{6.3.2}$$

gewählt. Abhängig von der Anzahl Elemente vergleichen wir Fehlerschätzer η_R und Fehler in Energienorm $||\nabla(u-U)||_{L^2(\Omega)}$ des adaptiven und uniformen Verfahrens. Dabei wird der Fehler mittels der Galerkin-Orthogonalität

$$||\nabla(u-U)||_{L^2(\Omega)} = \left(||\nabla u||^2_{L^2(\Omega)} - ||\nabla U||^2_{L^2(\Omega)}\right)^{1/2} \tag{6.3.3}$$

bestimmt. Die Norm $||\nabla U||^2_{L^2(\Omega)}$ der durch den Lösungsvektor $\mathbf{x}$ gegebenen diskreten Lösung U kann durch die zugehörige Steifigkeitsmatrix $\mathbf{A}$ mittels

$$||\nabla U||^2_{L^2(\Omega)} = \mathbf{x}\cdot\mathbf{A}\mathbf{x} \tag{6.3.4}$$

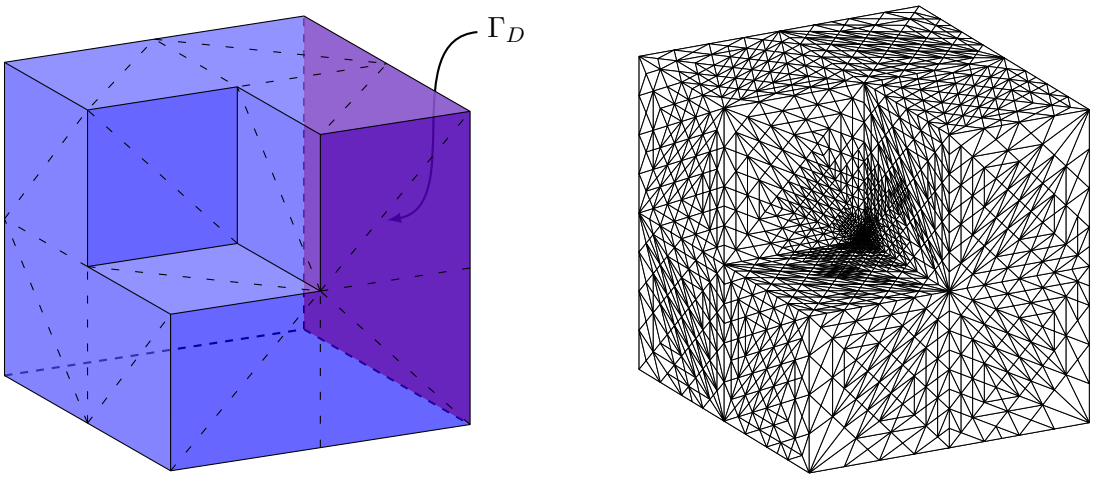


Abbildung 16: Triangulierung des sogenannten Fichera-Würfels: Links die Starttriangulierung $\mathcal{T}_0$ mit $\#\mathcal{T}_0 = 42$. Der Dirichlet-Rand $\Gamma_D = \{-1\} \times [-1, 1]^2$ ist rot markiert. Die übrigen blauen Randflächen stellen den Neumann-Rand $\Gamma \backslash \Gamma_D$ dar. Rechts ist die nach dreißig Schritten erhaltene adaptive Triangulierung zu sehen.

bestimmt werden. Die Norm der exakten Lösung wurde hierfür analytisch berechnet und beträgt ungefähr

$$||\nabla u||^2_{L^2(\Omega)} \approx 0.788687. \tag{6.3.5}$$

Den Abbildungen 17 und 18 sind nun sowohl die Fehler, als auch die Fehlerschätzer für uniforme Verfeinerung und adaptive Verfeinerung mit Adaptivitätsparameter $\rho = 0.25$ zu entnehmen. In Abbildung 17 sind für festen Quadraturgrad $q_{\text{solve}} = 4$ innerhalb der Funktion `solveLaplace` verschiedene Quadraturgrade q_{estimate} zur numerischen Integration innerhalb der Funktion `computeEtaR` verwendet worden. Der unterschiedliche Quadraturgrad ändert hier nur wenig am tatsächlichen Fehler, ein erhöhter Quadraturgrad zeigt allerdings für den Fehlerschätzer schneller das bessere Konvergenzverhalten des adaptiven Verfahrens an. Anders in Abbildung 18: Hier wird der Quadraturgrad des Fehlerschätzers festgehalten und der Quadraturgrad in der Funktion `solveLaplace` verändert. Durch höheren Quadraturgrad verringert sich hier vor allem der Fehler der uniformen Lösung merklich. Unabhängig von der gewählten Quadraturformel ist der Einsatz des adaptiven Verfahrens die sichtlich bessere Alternative. Das adaptive Verfahren liefert schon mit knapp 50.000 Elementen eine gleich gute Lösung, wie das uniforme Verfahren mit etwa einer halben Million Elemente.

Wir wollen nun den Vorteil des adaptiven Verfahrens etwas relativieren. Denn es darf nicht vergessen werden, dass zur Bestimmung der adaptiven Lösung wesentlich mehr Rechenschritte benötigt werden. Für das uniforme Verfahren muss nur bis zum gewünschten Grad verfeinert und anschließend einmal die Lösung bestimmt werden, wohingegen beim adaptiven Verfahren in jedem Schritt sowohl die Lösung, als auch die Fehlerindikatoren berechnet werden müssen. Wir vergleichen also in Abbildung 19 die Fehler und Fehlerschätzer in Abhängigkeit der aufgewandten Zeit. Doch auch hier zeichnet sich das adaptive Verfahren – wenn auch etwas später – klar vom uniformen ab.

Es zeigt sich also, dass das adaptive Verfahren nicht nur im Sinne der Anzahl Elemente, sondern auch im Sinne des Zeitaufwands rascher eine gute Lösung liefert als das uniforme.

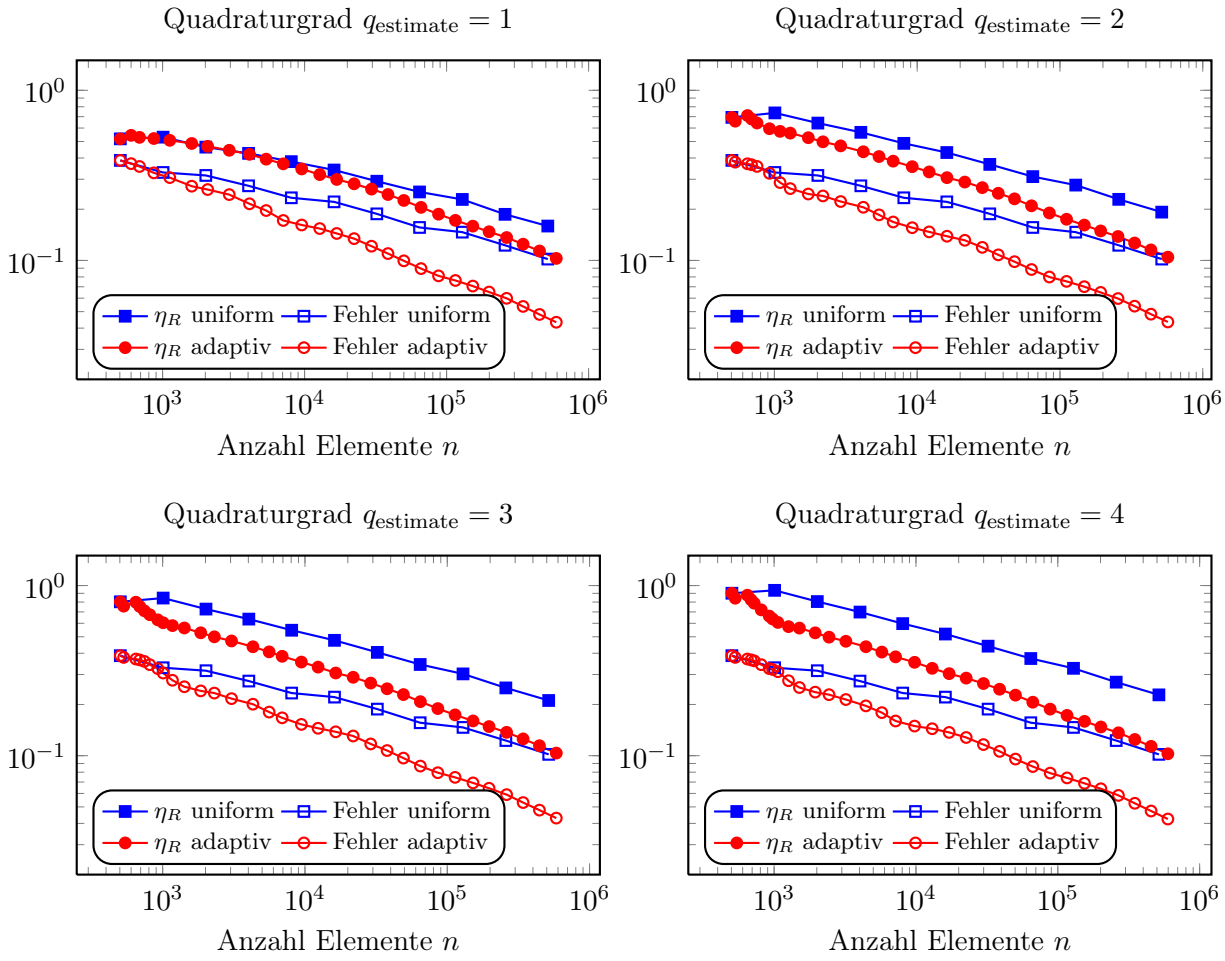


Abbildung 17: Numerische Ergebnisse für den Fehlerschätzer η_R und den tatsächlichen Fehler in Energienorm bei festem Quadraturgrad $q_{\text{solve}} = 4$ innerhalb der Funktion `solveLaplace` und variablem Quadraturgrad in der Funktion `computeEtaR`.

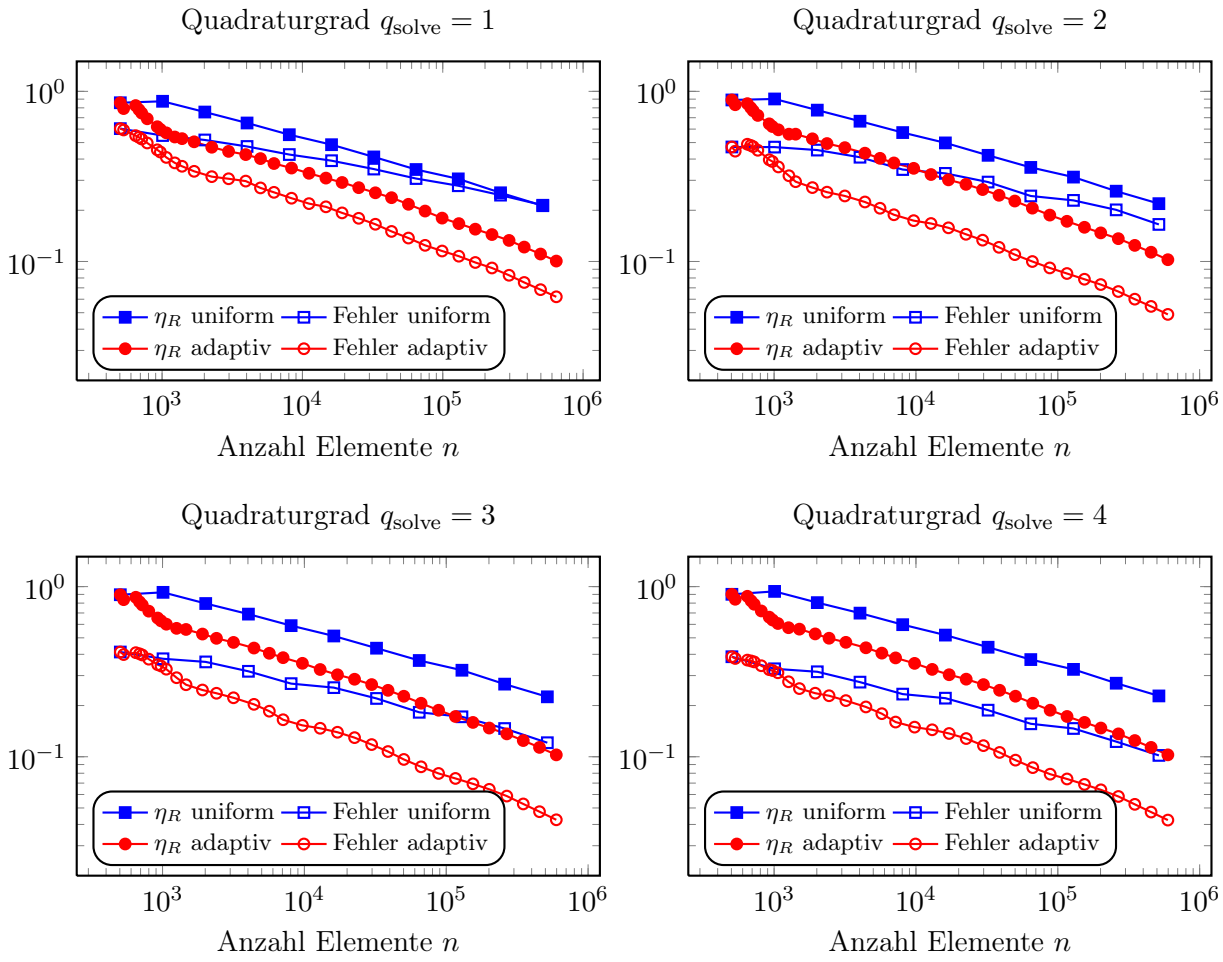


Abbildung 18: Numerische Ergebnisse für den Fehlerschätzer η_R und den tatsächlichen Fehler in Energienorm bei festem Quadraturgrad $q_{\text{estimate}} = 4$ innerhalb der Funktion `computeEtaR` und variablem Quadraturgrad in der Funktion `solveLaplace`.

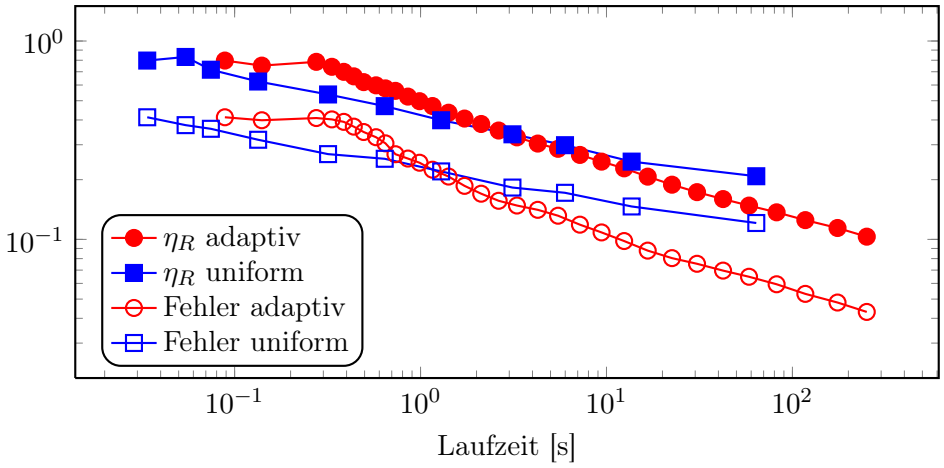


Abbildung 19: Fehlerschätzer η_R und tatsächlicher Fehler in Energienorm bei Quadraturgrad $q_{\text{estimate}} = q_{\text{solve}} = 3$ in Abhängigkeit der aufgewandten Zeit.

A Bisektionsverfahren in C

Listing 17: MATLAB-Wrapperfunktion für C-Verfeinerung

```
function [elements,elementgeneration,coordinates,varargout] = ...
        refineC(elements,elementgeneration,coordinates,varargin)
%*** refineC dient als Wrapper-Funktion, die die noetigen Hilfsstrukturen erzeugt.
%*** Die eigentliche Verfeinerung geschieht in der mex-Funktion refrekC
nB = length(varargin)-1;
if (nB~=0)
    [boundaries{1:nB}] = varargin{1:nB};
else
    boundaries = cell(1,1);
end
markedElements = unique(varargin{end});
if (size(elements,2)>4 || size(elementgeneration,2)~=1 || size(coordinates,2)~=3)
    error('Falsche Formate');
end
if (isempty(markedElements) || any(markedElements<0) || ...
                               any(markedElements>size(elements,1)) )
    error('markedElements-Format nicht korrekt');
end
%*** Generiere Hilfsstrukturen
[face2nodes,element2faces,boundary2face{1:nB}] = ...
                            provideFaceData(elements,boundaries{1:nB});
nF = size(face2nodes,1);
element2bdNumber = cell(1,nB);
for j = 1:nB
    nBE(j) = size(boundaries{j},1);
    bdNumber = zeros(nF,1);
    bdNumber(boundary2face{j}) = 1:nBE(j);
    element2bdNumber{j} = reshape(bdNumber(element2faces),[],4);
end
element2neigh = provideNeighbors(elements);
%*** Rufe MEX-Funktion
if (nB==0)
    [elements,elementgeneration,coordinates] = refrekC(elements,...
        elementgeneration,coordinates,element2neigh,markedElements);
else
    [elements,elementgeneration,coordinates,varargout{1:nB}] = refrekC(elements,...
            elementgeneration,coordinates,element2neigh,...
            boundaries{1:nB},element2bdNumber{1:nB},markedElements);
end
```

Listing 18: MEX-Code: mexFunction für Bisektionsverfahren

```
#include "mex.h"
#include "structs.c"
#include "buildMesh.c"
#include "divideSimplex.c"
#include "refine.c"
#include "convertToMatlab.c"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
    {
    int i;
    //### Element- und Koordinaten-Informationen
    double* elements = mxGetPr(prhs[0]);
    int nE = mxGetM(prhs[0]);
    int dim = mxGetN(prhs[0])-1;
    double* elementgeneration = mxGetPr(prhs[1]);
```

```
14    double* coordinates= mxGetPr(prhs[2]);
15    int nC = (int) mxGetM(prhs[2]);
16    double* elem2neigh = mxGetPr(prhs[3]);
17    //### Rand-Information
18    int nB = (nrhs-5)/2;
19    int* nBE = (int*) mxMalloc(nB*sizeof(int));
20    double** elem2bdNumber = (double**) mxMalloc(nB*sizeof(double*));
21    double** boundaries = (double**) mxMalloc(nB*sizeof(double*));
22    for (i=0;i<nB;i++){
23       boundaries[i] = mxGetPr(prhs[4+i]);
24       nBE[i] = mxGetM(prhs[4+i]);
25       elem2bdNumber[i] = mxGetPr(prhs[4+nB+i]);
26    }
27    //### Markierte Elemente
28    double* marked = mxGetPr(prhs[nrhs-1]);
29    int nM = mxGetM(prhs[nrhs-1])*mxGetN(prhs[nrhs-1]);
30    //### Aufbauen, Verfeinern, Zurueckgeben
31    mesh* M;
32    M = buildMesh(dim,nE,elements,elementgeneration,nC,coordinates,elem2neigh
          ,nB,nBE,boundaries,elem2bdNumber,nM,marked);
33    refine(M);
34    convertToMatlab(nlhs, plhs, M);
35 }
```

Listing 19: MEX-Code: structs.c

```
1  typedef struct node{
2     int num;
3     double co[3];
4     struct node* next;
5  } node;
6
7  typedef struct simplex{
8     int boundaryNum;/*boundarNum =
9     -1: Rand, aber nicht uebergeben worden,
10    0: kein Rand sondern Tetraeder,
11    1: boundary[0],
12    2: boundary[1], ...*/
13    int generation;
14    int dim;
15    node** node;
16    struct simplex** neighbor;
17    struct simplex* child0;
18    struct simplex* childn;
19    struct simplex* parent;
20    struct simplex* next;
21    struct simplex* before;
22 } simplex;
23
24 typedef struct mesh{
25    int dim;
26    int nE;
27    int nB;
28    int* nBE;
29    int nC;
30    node* firstNode;
31    node* lastNode;
32    simplex* firstElement;
33    simplex** firstBdElement;//#firstBdElement[i] pointer auf ersten boundary
          [i]
34    simplex** todo;
35    int todoSize;
36 } mesh;
37
38 node* newNode(double cx, double cy, double cz){
39    node* newnode = (node*)mxMalloc(sizeof(node));
40    newnode->co[0] = cx;
41    newnode->co[1] = cy;
42    newnode->co[2] = cz;
43    return newnode;
44 }
45
```

```
46 node* generateMidpoint(node* a, node* b){
47    return newNode((a->co[0]+b->co[0])/2,(a->co[1]+b->co[1])/2,(a->co[2]+b->
          co[2])/2);
48 }
49
50 simplex* newSimplex(simplex* parent, int generation){
51    simplex* newsimplex = (simplex*)mxMalloc(sizeof(simplex));
52    newsimplex->dim = 0;
53    newsimplex->boundaryNum = 0;
54    newsimplex->child0 = NULL;
55    newsimplex->childn = NULL;
56    newsimplex->neighbor = NULL;
57    newsimplex->parent = NULL;
58    newsimplex->next = NULL;
59    newsimplex->before = NULL;
60    if (parent!=NULL){
61       newsimplex->node=(node**)mxMalloc((parent->dim+1)*sizeof(node*));
62       newsimplex->parent=parent;
63       newsimplex->dim = parent->dim;
64       newsimplex->boundaryNum = parent->boundaryNum;
65       if (parent->neighbor!=NULL){//### Kein Randelement
66          newsimplex->neighbor = (simplex**) mxMalloc((newsimplex->dim+1)*
              sizeof(simplex*));
67       }
68    }
69    newsimplex->generation = generation;
70    return newsimplex;
71 }
72 simplex* genUndefBd(){
73    simplex* newsimplex = (simplex*) mxMalloc(sizeof(simplex));
74    newsimplex->dim = 0;
75    newsimplex->boundaryNum = -1;
76    newsimplex->generation = -1;
77    newsimplex->child0 = NULL;
78    newsimplex->childn = NULL;
79    newsimplex->parent = NULL;
80    newsimplex->neighbor = NULL;
81    newsimplex->next = NULL;
82    newsimplex->before = NULL;
83    return newsimplex;
84 }
85
86 int isundefBd(simplex* S){
87    if (S!=NULL){
88       return (S->boundaryNum==-1);
89    }else{
90       return 1;
91    }
92 }
93 int hasSameRefEdge(simplex* S, simplex* T){
94    if ((S->node[0]==T->node[0]) && (S->node[S->dim]==T->node[S->dim])){
95       return 1;
96    }
97    if ((S->node[0]==T->node[S->dim]) && (S->node[S->dim]==T->node[0])){
98       return 1;
99    }
100   return 0;
101 }
```

Listing 20: MEX-Code: buildMesh.c

```
1 //### buildMesh:
2 mesh* buildMesh(int dim, int nE, double* elements,double* elementgeneration,
       int nC, double* coordinates,double* elem2neigh,int nB,int*nBE, double**
      boundariesMAT,double** elem2bdNumber,int nM, double* marked){
3    int i,j,k;
4    mesh* M = (mesh*) mxMalloc(sizeof(mesh));
5    simplex* simplices;
6    simplex** boundaries;
7    simplex* undefBd = genUndefBd();//### Dummy-Rand, um haeufige NULL-
        Abfragen zu vermeiden.
8    node* nodes;
```

```
9
10    M->dim = dim;
11    M->nE = nE;
12    M->nB = nB;
13    M->nBE = (int*) mxMalloc(nB*sizeof(int));
14    for (i=0;i<nB;i++){
15       M->nBE[i] = nBE[i];
16    }
17    M->nC = nC;
18    //### Knoten anlegen
19    nodes = (node*) mxMalloc(nC*sizeof(node));
20    for (i=0;i<nC;i++){
21       (&nodes[i])->co[0] = coordinates[i];
22       (&nodes[i])->co[1] = coordinates[i+nC];
23       (&nodes[i])->co[2] = coordinates[i+2*nC];
24    }
25    for (i=0;i<nC-1;i++){
26       (&nodes[i])->num = i+1;
27       (&nodes[i])->next = &(nodes[i+1]);
28    }
29    nodes[nC-1].num = nC;
30    nodes[nC-1].next = NULL;
31    M->firstNode = &(nodes[0]);
32    M->lastNode = &(nodes[nC-1]);
33    //### Simplizes anlegen
34    simplices = (simplex*) mxMalloc(nE*sizeof(simplex));
35
36    for (i=0;i<nE;i++){
37       simplices[i].boundaryNum = 0;
38       simplices[i].generation = elementgeneration[i];
39       simplices[i].dim = dim;
40       simplices[i].child0 = NULL;
41       simplices[i].childn = NULL;
42       simplices[i].parent = NULL;
43       simplices[i].neighbor = (simplex**) mxMalloc((dim+1)*sizeof(simplex*))
             ;
44       simplices[i].node = (node**) mxMalloc((dim+1)*sizeof(node*));
45       for (j=0;j<dim+1;j++){
46          int nachb = (int)elem2neigh[j*nE+i];
47          if (nachb!=0){
48             simplices[i].neighbor[j] = &(simplices[nachb-1]);
49          }else{
50             simplices[i].neighbor[j] = undefBd;
51          }
52          simplices[i].node[j] = &(nodes[(int)elements[j*nE+i]-1]);
53       }
54    }
55    simplices[0].before = NULL;
56    if (nE>1){
57       simplices[0].next = &(simplices[1]);
58    }
59    for (i=1;i<nE-1;i++){
60       simplices[i].next = &(simplices[i+1]);
61       simplices[i].before = &(simplices[i-1]);
62    }
63    if (nE>1){
64       simplices[nE-1].before = &(simplices[nE-2]);
65    }
66    simplices[nE-1].next = NULL;
67    M->firstElement = &(simplices[0]);
68    //### Raender anlegen
69    if (nB!=0){
70       boundaries = (simplex**) mxMalloc(nB*sizeof(simplex*));
71       M->firstBdElement = (simplex**) mxMalloc(nB*sizeof(simplex*));
72       for (k=0;k<nB;k++){
73          boundaries[k] = (simplex*) mxMalloc(nBE[k]*sizeof(simplex));
74          for (i=0;i<nBE[k];i++){
75             boundaries[k][i].boundaryNum = k+1;
76             boundaries[k][i].generation = 0;
77             boundaries[k][i].dim = dim-1;
78             boundaries[k][i].child0 = NULL;
79             boundaries[k][i].childn = NULL;
80             boundaries[k][i].parent = NULL;
81             boundaries[k][i].neighbor = NULL;
```

```
82              boundaries[k][i].node = (node**) mxMalloc(dim*sizeof(node*));
83              for (j=0;j<dim;j++){
84                  boundaries[k][i].node[j] = &(nodes[(int)boundariesMAT[k][j*
                        nBE[k]+i]-1]);
85              }
86          }
87          boundaries[k][0].before = NULL;
88          if (nBE[k]>1){
89              boundaries[k][0].next = &(boundaries[k][1]);
90          }
91          for (i=1;i<nBE[k]-1;i++){
92              boundaries[k][i].next = &(boundaries[k][i+1]);
93              boundaries[k][i].before = &(boundaries[k][i-1]);
94          }
95          if (nBE[k]>1){
96              boundaries[k][nBE[k]-1].before = &(boundaries[k][nBE[k]-2]);
97          }
98          boundaries[k][nBE[k]-1].next = NULL;
99
100
101         M->firstBdElement[k] = boundaries[k];
102         //### Nachbarschaftsrelationen auf boundaries
103         for(i=0;i<nE;i++){
104             for(j=0;j<dim+1;j++){
105                 int randflaeche = (int) elem2bdNumber[k][i+j*nE];
106                 if (randflaeche!=0){
107                     simplices[i].neighbor[j] = &(boundaries[k][randflaeche-1])
                            ;
108                 }
109             }
110         }
111     }
112 }
113 M->todoSize = nM;
114 M->todo = (simplex**) mxMalloc(nM*sizeof(simplex*));
115 for (i=0;i<nM;i++){
116     M->todo[i] = &(simplices[(int)marked[i]-1]);
117 }
118
119 return M;
120 }
```

Listing 21: MEX-Code: refine.c

```
1  //### Refine:
2  void refine(mesh* M){
3      simplex* simpl;
4      int i,k;
5      for (i=0;i<M->todoSize;i++){
6          divideSimplex(M->todo[i],NULL,M);
7      }
8      //### Neues firstElement finden
9      simpl = M->firstElement;
10     while (simpl->child0!=NULL){
11         simpl = simpl->child0;
12     }
13     M->firstElement = simpl;
14     //### neue firstBdElements finden
15     for (k=0;k<M->nB;k++){
16         simpl=M->firstBdElement[k];
17         while (simpl->child0!=NULL){
18             simpl = simpl->child0;
19         }
20         M->firstBdElement[k] = simpl;
21     }
22 }
```

Listing 22: MEX-Code: divideSimplex.c

```
1  //### divideSimplex:
2  //### S wird geteilt, Aufruf durch T
3  void divideSimplex (simplex* S, simplex* T, mesh* M){
4     int n = S->dim;
5     int i,p;
6     int gamma;
7     node* newnode;
8     if (S->child0 != NULL || isundefBd(S)){//### Kein Teilen noetig
9        return;
10    }
11    if (n == M->dim){//### Randflaechen duerfen keine Teilung initiieren
12       for (i=1;i<n;i++){
13          if (S->neighbor[i]->dim == M->dim){//### Randflaechen benoetigen
                keine Mehrfachteilung
14             if (S->neighbor[i]->generation < S->generation){
15                divideSimplex(S->neighbor[i],NULL,M);
16             }
17          }
18       }
19    }
20    if (T==NULL){//### Erster Tetraeder des Edgepatches
21       newnode = generateMidpoint(S->node[0],S->node[n]);
22       newnode->num = M->lastNode->num+1;
23       newnode->next = NULL;
24       M->lastNode->next = newnode;
25       M->lastNode = newnode;
26       M->nC = M->nC+1;
27    }else{
28       newnode = T->child0->node[1];
29    }
30    S->child0 = newSimplex(S,S->generation+1);
31    S->childn = newSimplex(S,S->generation+1);
32    if (S->boundaryNum == 0){
33       (M->nE)++;
34    }else{
35       (M->nBE[S->boundaryNum-1])++;
36    }
37    //### Verkettete Liste aktualisieren
38    if (S->before != NULL){
39       S->before->next = S->child0;
40    }
41    S->child0->before = S->before;
42    S->child0->next = S->childn;
43    S->childn->before = S->child0;
44    S->childn->next = S->next;
45    if(S->next!=NULL){
46       S->next->before = S->childn;
47    }
48    //### Knoten anpassen
49    S->child0->node[0] = S->node[0];
50    S->childn->node[0] = S->node[n];
51
52    S->child0->node[1] = newnode;
53    S->childn->node[1] = newnode;
54
55    gamma = (S->generation)%n;
56    for (p=1;p<n;p++){
57       int pn=p+1;
58       int p0=p+1;
59       if (p>gamma){
60          pn = n-p+gamma+1;
61       }
62       S->childn->node[pn] = S->node[p];
63       S->child0->node[p0] = S->node[p];
64       if (n == M->dim){
65          if (S->neighbor[p]->child0 != NULL){
66             if (S->neighbor[p]->node[0]==S->node[0]){
67                S->child0->neighbor[p0] = S->neighbor[p]->child0;
68                S->childn->neighbor[pn] = S->neighbor[p]->childn;
69             }else{
70                S->child0->neighbor[p0] = S->neighbor[p]->childn;
71                S->childn->neighbor[pn] = S->neighbor[p]->child0;
72             }
73          }else{
```

```
74              S->childn->neighbor[pn] = S->neighbor[p];
75              S->child0->neighbor[p0] = S->neighbor[p];
76           }
77        }
78     }
79     if (n == M->dim){//### Falls Rand, bleibt neighbor unangetastet
80        S->child0->neighbor[0] = S->childn;
81        S->childn->neighbor[0] = S->child0;
82
83        if (S->neighbor[n]->child0 != NULL){
84           if (hasSameRefEdge(S->child0,S->neighbor[n]->child0)){//### S->
               child0 passt mit S->neighbor[n]->child0 zusammen
85              S->child0->neighbor[1] = S->neighbor[n]->child0;
86           }else{//### S->child0 passt mit S->neighbor[n]->childn zusammen
87              S->child0->neighbor[1] = S->neighbor[n]->childn;
88           }
89        }else{
90           S->child0->neighbor[1] = S->neighbor[n];
91        }
92        if (S->neighbor[0]->child0 != NULL){
93           if (hasSameRefEdge(S->childn,S->neighbor[0]->childn)){//### S->
               childn passt mit S->neighbor[0]->childn zusammen
94              S->childn->neighbor[1] = S->neighbor[0]->childn;
95           }else{//### S passt mit S->neighbor->child0 zusammen
96              S->childn->neighbor[1] = S->neighbor[0]->child0;
97           }
98        }else{
99           S->childn->neighbor[1] = S->neighbor[0];
100       }
101       for (i=0;i<M->dim+1;i++){
102          if (S->child0->neighbor[i]->generation==S->child0->generation){
103             S->child0->neighbor[i]->neighbor[i] = S->child0;
104          }
105          if (S->childn->neighbor[i]->generation==S->childn->generation){
106             S->childn->neighbor[i]->neighbor[i] = S->childn;
107          }
108       }
109       for (i=1;i<n;i++){
110          divideSimplex(S->neighbor[i],S,M);
111       }
112    }else{//### Hier ist S Randflaeche und wurde von T aufgerufen
113       // Die Kinder von T muessen auf soeben erstellte Randflaechen
             verweisen.
114       for (i=0;i<M->dim+1;i++){
115          if (T->child0->neighbor[i]==S){
116             if (T->child0->node[0]==S->node[0])
117                T->child0->neighbor[i] = S->child0;
118             else
119                T->child0->neighbor[i] = S->childn;
120          }
121          if (T->childn->neighbor[i]==S){
122             if (T->childn->node[0]==S->node[0])
123                T->childn->neighbor[i] = S->child0;
124             else
125                T->childn->neighbor[i] = S->childn;
126          }
127       }
128    }
129 }
```

Listing 23: MEX-Code: convertToMatlab.c

```
1  //### convertToMatlab
2  void convertToMatlab(int nlhs, mxArray *plhs[], mesh* M){
3     int k,i,j;
4     double* elements;
5     double* elementgeneration;
6     double* coordinates;
7     double* boundary;
8     simplex* aktsimplex;
9     simplex* aktboundary;
10    node* aktnode;
```

```
11      //### elements
12      if(nlhs>=1){
13         plhs[0] = mxCreateDoubleMatrix(M->nE,M->dim+1,mxREAL);
14         elements = mxGetPr(plhs[0]);
15         //
16         i = 0;
17         for(j=0;j<M->dim+1;j++){
18            aktsimplex = M->firstElement;
19            while(aktsimplex!=NULL){
20               elements[i] = aktsimplex->node[j]->num;
21               aktsimplex = aktsimplex->next;
22               i++;
23            }
24         }
25      }
26      //### elementgeneration
27      if(nlhs>=2){
28         plhs[1] = mxCreateDoubleMatrix(M->nE,1,mxREAL);
29         elementgeneration = mxGetPr(plhs[1]);
30         aktsimplex = M->firstElement;
31         i=0;
32         while(aktsimplex!=NULL){
33            elementgeneration[i] = aktsimplex->generation;
34            aktsimplex = aktsimplex->next;
35            i++;
36         }
37      }
38      //### coordinates
39      if(nlhs>=3){
40         plhs[2] = mxCreateDoubleMatrix(M->nC,3,mxREAL);
41         coordinates = mxGetPr(plhs[2]);
42         i=0;
43         for(j=0;j<3;j++){
44            aktnode = M->firstNode;
45            while(aktnode!=NULL){
46               coordinates[i] = aktnode->co[j];
47               aktnode = aktnode->next;
48               i++;
49            }
50         }
51      }
52      //### boundaries
53      for (k=0;k<M->nB;k++){
54         if(nlhs>=4+k){
55            plhs[3+k] = mxCreateDoubleMatrix(M->nBE[k],M->dim,mxREAL);
56            boundary = mxGetPr(plhs[3+k]);
57            i = 0;
58            for(j=0;j<M->dim;j++){
59               aktboundary = M->firstBdElement[k];
60               while(aktboundary!=NULL){
61                  boundary[i] = aktboundary->node[j]->num;
62                  aktboundary = aktboundary->next;
63                  i++;
64               }
65            }
66         }
67      }
68   }
```

B Übersicht der implementierten Funktionen

An das Ende sei noch einmal ein Überblick über die entwickelten Funktionen gestellt. Um die Notation zu verkürzen, schreiben wir `C (coordinates)`, `E (elements)`, `G (elementgeneration)`, `D (dirichlet)` und `N (neumann)`. Von den MATLAB-Funktionen f und g setzen wir voraus, dass sie Punkte $\xi_j \in \mathbb{R}^3$ in Form einer Matrix $\xi \in \mathbb{R}^{n\times 3}$ übernehmen und den zugehörigen Vektor $y \in \mathbb{R}^n$ zurückliefern, für den gilt $y_j = f(\xi_j)$ bzw. $y_j = g(\xi_j)$.

B.1 Lösung der Laplace-Gleichung in 3D

Der vektorisierte Solver `solveLaplace` wird durch

```
[x,energy] = solveLaplace(C,E,D,N,f,g,uD);
```
(B.1.1)

aufgerufen. Die Funktionen `solveLaplace0` und `solveLaplace1` haben die gleiche Signatur.

B.2 Lokale Netzverfeinerung mittels Bisektion

Die Funktion zur Netzverfeinerung mittels Bisektion wird durch

```
[E,G,C,D,N] = refineC(E,G,C,D,N,marked);
```
(B.2.1)

aufgerufen, wobei `marked` einen Vektor mit den Indizes der markierten Elemente darstellt. Das Eingabenetz muss dabei für das Bisektionsverfahren zulässig sein. Dies ist beispielsweise der Fall, wenn es mittels `validStartMesh` aus einer regulären Triangulierung generiert wurde, oder wiederum daraus mittels Bisektion hervorgegangen ist. Das Ergebnis ist eine reguläre Triangulierung, sodass die markierten Tetraeder verfeinert wurden und sie weiterhin für das Bisektionsverfahren geeignet ist. Die Funktion `refine` ist eine ineffiziente Implementierung in MATLAB und hat die gleiche Signatur.

B.3 Erzeugung einer gültigen Starttriangulierung

Um aus einer regulären Triangulierung eine für Bisektion passende Starttriangulierung zu erzeugen wird der Befehl

```
[E,G,C,D,N] = validStartMesh(E,C,D,N);
```
(B.3.1)

verwendet.

B.4 Vergröbern einer mittels Bisektion erhaltenen Triangulierung

Triangulierungen, die mittels Bisektion generiert wurden können mit dem Befehl

```
[E,G,C,D,N] = coarsen(E,G,C,D,N,marked);
```
(B.4.1)

vergröbert werden, wobei `marked` einen Vektor mit den Indizes der zu vergröbernden Tetraeder darstellt. Die Rückgabe ist dabei wieder für Bisektion zulässig. Um dies zu gewährleisten werden hierbei eventuell nicht alle Elemente aus `marked` vergröbert, jedenfalls aber die größtmögliche Teilmenge von `marked`, sodass die resultierende Triangulierung wieder für Bisektion zulässig ist.

B.5 Residuum-basierter Fehlerschätzer

Den Vektor der Fehlerindikatoren erhält man durch den Aufruf

```
etaR = computeEtaR(x,E,C,D,N,f,g);
```

Der Vektor `x` ist dabei der Koeffizientenvektor der mittels `solveLaplace` generierten Lösung.

B.6 Adaptive Netzverfeinerung

Das adaptive Verfahren lässt sich durch den Befehl

```
[x,C,E,etaR] = adaptiveAlgorithm(E,G,C,D,N,f,g,uD,nEmax,rho);
```
(B.6.1)

ausführen. Der Parameter `rho` $\in (0,1)$ ist ein vorgegebener Parameter der Adaptivität und `nEmax` die maximale Anzahl der Elemente. Zurückgegeben wird neben der Triangulierung und dem Koeffizientenvektor der zugehörigen Lösung auch der Vektor der Fehlerindikatoren `etaR`.

B.7 Rotverfeinerung

Die Rotverfeinerung wird mit dem Befehl

```
[E,C,D,N] = redRefine(E,C,D,N);
```
(B.7.1)

durchgeführt. Aus einer regulären Triangulierung wird hier durch Halbierung aller Kanten eine weitere reguläre Triangulierung erzeugt.

B.8 Quadratur auf Tetraedern und Dreiecken

Häufig werden in obigen Funktionen Quadraturregeln benötigt. Die Funktionen `tetquad` und `triquad` liefern hierfür geeignete Quadraturpunkte und zugehörige Gewichte. Die Funktionsaufrufe erfolgen mittels

```
[XYZ,W,sizeT,V] = tetquad(n,coord);
```
(B.8.1)

beziehungsweise

```
[XYZ,W,sizeT,V] = triquad(n,coord);
```
(B.8.2)

Dabei ist `n` der gewünschte Quadraturgrad je Dimension und `coord` ein $(d+1)\times 3$-Array der Eckpunktkoordinaten. Zurückgeliefert werden neben den Quadraturpunkten `XYZ` und -gewichten `W` auch das Volumen bzw. die Fläche des Elements `sizeT`, sowie die Quadraturpunkte in baryzentrischen Koordinaten `V`, welche auch als Funktionswerte der Hutfunktionen in den Quadraturpunkten interpretiert werden können. Quadratur der Funkton `f` kann dann mittels `I = W*f(XYZ)` erfolgen.

B.9 Bestimmung geometrischer Relationen

Die zur Verfeinerung und zur Berechnung der Fehlerindikatoren benötigten geometrischen Informationen werden durch die drei Funktionen `provideNeighbors`, `provideFaceData` und `provideEdgeData` zur Verfügung gestellt. Nachbarschaftsrelationen erhält man durch Aufruf von

```
element2neighbors = provideNeighbors(E);
```
(B.9.1)

Um Informationen über die Flächen der Triangulierung zu erhalten wird

```
[face2nodes,element2faces,dirichlet2face,neumann2face] = ...
                                        provideFaceData(E,D,N);
```
(B.9.2)

aufgerufen. Der Funktionsaufruf von `provideEdgeData` erfolgt mittels

```
[edge2nodes,element2edges,dirichlet2edges,neumann2edges] = ...
                                        provideEdgeData(E,D,N);
```
(B.9.3)

Literatur

[1] Alberty, Jochen, Carsten Carstensen und Stefan A. Funken: *Remarks Around 50 Lines Of Matlab: Short Finite Element Implementation.* Numerical Algorithms, 20:117–137, 1998.

[2] Bartels, Sören und Patrick Schreier: *Local coarsening of triangulations created by bisections.* Preprint, Universität Bonn, 2010.

[3] Bey, Jürgen: *Simplicial grid refinement: on Freudenthal's algorithm and the optimal number of congruence classes.* Numer. Math., 85(1):1–29, 2000.

[4] Bey, Jürgen: *Tetrahedral grid refinement.* Computing, 55(4):355–378, 1995.

[5] Cascon, Juan Manuel, Christian Kreuzer, Ricardo H. Nochetto und Kunibert G. Siebert: *Quasi-optimal convergence rate for an adaptive finite element method.* SIAM J. Numer. Anal., 46(5):2524–2550, 2008.

[6] Funken, Stefan, Dirk Praetorius und Philipp Wissgott: *Efficient Implementation of Adaptive P1-FEM in MATLAB.* Computational Methods in Applied Mathematics, 11(4):460 – 490, 2011.

[7] Kossaczký, Igor: *A recursive approach to local mesh refinement in two and three dimensions.* J. Comput. Appl. Math., 55(3):275–288, 1994.

[8] Königsberger, Konrad: *Analysis 2, 5. Auflage.* Springer-Verlag, Berlin, 2009.

[9] Maubach, Joseph Maria: *Local bisection refinement for n-simplicial grids generated by reflection.* SIAM J. Sci. Comput., 16(1):210–227, 1995.

[10] Praetorius, Dirk: *Finite Element Methode.* Vorlesungsskript, TU Wien, 2010.

[11] Praetorius, Dirk: *Numerische Mathematik.* Vorlesungsskript, TU Wien, 2010.

[12] Rahman, Talal und Jan Valdman: *Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements.* Technischer Bericht, Universität Leipzig, 2011.

[13] Rivara, María Cecilia: *Local modification of meshes for adaptive and/or multigrid finite-element methods.* J. Comput. Appl. Math., 36(1):79–89, 1991.

[14] Stevenson, Rob: *The completion of locally refined simplicial partitions created by bisection.* Math. Comp., 77(261):227–241, 2008.

[15] Traxler, Christoph: *An algorithm for adaptive mesh refinement in n dimensions.* Computing, 59(2):115–137, 1997.

[16] Verfürth, Rüdiger: *A review of a posteriori error estimation and adaptive mesh-refinement techniques.* Wiley-Teubner series in advances in numerical mathematics. Wiley-Teubner, New York, 1996.

[17] Weiss, Kenneth und Leila De Floriani: *Diamond Hierarchies of Arbitrary Dimension.* Computer Graphics Forum (Proceedings SGP 2009), 28(5):1289–1300, 2009.

[18] Weiss, Kenneth und Leila De Floriani: *Bisection-based triangulations of nested hypercubic meshes.* In: Shontz, S. (Herausgeber): *Proceedings 19th International Meshing Roundtable*, Seiten 315–333, Chattanooga, Tennessee, October 3–6 2010.

[19] Weiss, Kenneth und Leila De Floriani: *Simplex and Diamond Hierarchies: Models and Applications.* Accepted to Computer Graphics Forum, 2011.

[20] Zhang, Shangyou: *Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes.* Houston J. Math., 21(3):541–556, 1995.