Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Orthogonale Dünngitter-Teilraumzerlegungen

Constantin Schreiber

Studiengang:	Informatik
Prüfer/in:	JP Dr. Dirk Pflüger
Betreuer/in:	Dr. Stefan Zimmer
Beginn am:	14. September 2017
Beendet am:	14. März 2018

Abstract

In der Simulation treten Häufig hochdimensionale partielle Differentialgleichungen auf. Das Lösen dieser wird für volle Gitter sehr schnell zu teuer. In dieser Arbeit wird ein Verfahren für das Lösen partieller Differentialgleichungen mit Hilfe von Dünnen Gittern, welche für mehrdimensionale Probleme besser skalieren, sowie dessen Implementierung in das Programmpaket SG++ vorgestellt. Durch Funktionsdarstellung in einem Erzeugendensystem wird die Verwendung einer L2-orthogonalen Teilraumzerlegung ermöglicht. Projektionsoperatoren ersetzen hierbei die explizite Transformation in eine Prewavelet-Basis. Diese Zerlegung erlaubt das Lumping der Steifigkeitsmatrix, also das Weglassen von großen Blöcken der Matrix. Hiermit wird ein Algorithmus zur Matrixmultiplikation, welcher dem von Schwab und Todor ähnelt implementiert. Dieser wird in einem konjugierten Gradienten-Verfahren verwendet und auch auf krummberandete Gebieten angewendet. Des Weiteren wird die Teilraumzerlegung durch L2-Projektion mit anderen Zerlegungen in Bezug auf Laufzeit und Fehlerentwicklung verglichen.

Inhaltsverzeichnis

1 Einleitung

Der Finite-Elemente-Ansatz, der in der Simulation häufig verwendet wird um partielle Differentialgleichungen zu diskretisieren und zu lösen kann mithilfe des sogenannten Galerkin-Ansatzes in das Lösen eines linearen Gleichungssystems überführt werden. Wenn hiermit allerdings hochdimensionale Probleme angegangen werden sollen, wie dies häufig der Fall ist, führen volle Gitter mit N Gitterpunkten pro Dimension und d Dimensionen zu einer Matrix, welche sich in der Größenordnung $O(N^{2d}))$ befindet. Diese zu lösen ist zu teuer, weshalb der Ansatz der dünnen Gitter verwendet wird. Hierbei werden Stützpunkte ignoriert, welche nur geringe Beiträge zur Approximationsgüte liefern. Somit verringert sich die Anzahl der Gitterpunkte auf $O(N(\log N)^{d-1})$, ohne dass ein großer Fehler eingeführt wird.

Um das lineare Gleichungssystem zu lösen bietet sich beispielsweise das Verfahren der konjugierten Gradienten an. Um dieses anzuwenden muss eine effiziente Multiplikation mit der sogenannten Steifigkeitsmatrix, welche der Differentialoperation definiert, implementiert werden. Hierfür müssen einzelne Teile der dünnen Gitter mit Blöcken der Matrix multipliziert werden. Wenn der Differentialoperator keine Tensorproduktstruktur besitzt, müsste hierbei ein Transport über zu feine Level erfolgen. Deshalb wird in [RP] eine Methode vorgestellt, welche Prewavelets als Basisfunktion ausnutzt. Diese besitzen die nützliche Eigenschaft der L_2-Orthogonalität, was dazu führt, dass der Transport über die Level, welche nicht im dünnen Gitter enthalten sind weggelassen werden kann.

In dieser Arbeit soll ein Algorithmus implementiert werden, welcher auch die Orthogonalität ausnutzt, ohne allerdings explizit in die Prewavelet-Basis zu transformieren. Dies führt zu einer übersichtlicheren Darstellung des Algorithmus und dazu, dass der Algorithmus sich leicht auf andere Operatoren und höhere Dimensionen anwenden lassen sollte. Durch Darstellung der Funktionen in einem Erzeugendensystem wird eine L_2-Projektion zwischen den Levels ermöglicht, welche die benötigte Orthogonalität erzeugt.

Die Arbeit ist in 5 Teile gegliedert, zu Beginn werden die benötigten mathematischen Grundlagen erläutert. Anschließend wird der Algorithmus, und alles was für ihn benötigt ist, beschrieben. Dann wird genauer auf das Framework SG++ und die Implementierung in diesem eingegangen und die dazugehörigen Resultate vorgestellt. Zum Abschluss gibt es noch einen Ausblick auf mögliche Anpassungen des Algorithmus.

2 Mathematische Grundlagen

2.1 Finite Elemente

Ein häufig verwendeter Ansatz um partielle Differentialgleichungen numerisch zu lösen ist das Finite-Elemente Verfahren. Um die Randwertaufgabe zu diskretisieren wird hier der sogenannte Galerkin-Ansatz nach [Hac96] verwendet. Dieser Ansatz basiert auf der variationellen (schwachen) Darstellung des Problems:

(2.1) suche $u \in V : a(u,v) = f(v) \ \ \forall v \in V$

mit der elliptischen Bilinearform

$$a(u,v) = \int \nabla u D(\nabla v)^T + \nabla u \underline{b} v + ucv \, d\underline{x}$$

und der rechten Seite

$$f(v) = \int fv \, d\underline{x}$$

Im Hauptteil dieser Arbeit wird $\underline{b}$ und c auf 0 gesetzt, was zu einer symmetrischen Bilinearform führt. Der unendlich-dimensionale Raum V wird nun durch einen diskreten Unterraum

$$V_N \subset V \text{ mit } \dim V_N = N < \infty$$

ersetzt. Somit ergibt sich aus 2.1 als Aufgabe

(2.2) suche $u^N \in V_N : a(u^N, v) = f(v) \ \ \forall v \in V_N$

Sei nun $B_N = \varphi_1, \ldots, \varphi_N$ eine Basis von V_N, also

$$\text{span}\{\varphi_1, \ldots, \varphi_N\} = V_N$$

Mit dieser Basis ist 2.2 äquivalent zu

$$\text{suche } u^N \in V_N : a(u^N, \varphi_i) = f(\varphi_i) \ \text{ für } i = 1, \ldots, N$$

Wenn nun auch noch u^N als Kombination von Basisvektoren ausgedrückt und der Koeffizient aus der Bilinearform gezogen wird, ergibt sich:

$$\text{suche } \underline{u} : \sum_{j=1}^{N} u_j a(\varphi_j, \varphi_i) = f(\varphi_i) \ \text{ für } i = 1, \ldots, N$$

Nun lässt sich dieses Problem auch in Matrixform umschreiben:

(2.3) suche $\underline{u}: \ A\underline{u} = \underline{f}$

Hierbei ist A, die sogenannte Steifigkeitsmatrix, folgendermaßen definiert:

$A_{i,j} = a(\varphi_i, \varphi_j)$

Und der Vektor $\underline{f}$ als

$f_i = f(\varphi_i)$

Das Problem die partielle Differentialgleichung zu lösen wurde somit auf ein lineares Gleichungssystem reduziert. Der Rest dieser Arbeit widmet sich nun dem Lösen dieses Gleichungssystems.

2.1.1 Krummberandete Gebiete

In der Simulation treten häufig Probleme auf krummberandeten Gebieten auf. Um auch diese Anwendungsfälle bearbeiten zu können gibt es mehrere Möglichkeiten. Für diese Arbeit wird ein Transformationsansatz aus [Dor97] gewählt, das bedeutet, dass das Problemgebiet auf das Einheitsintervall zurückgeführt wird. Dies erleichtert den Aufbau des Algorithmus, da sich im Vergleich zu der ursprünglichen Aufgabe nur der Differentialoperator ändert, während der restliche Ablauf unverändert bleibt. Sei die Abbildung zwischen Einheits- und physikalischem Gebiet wie folgt definiert:

$\psi : \mathbf{Q} \to \mathbf{P}, \ \underline{x} \mapsto \underline{\xi} = \underline{\psi}(\underline{x})$

Um den neuen Differentialoperator zu berechnen wird die mehrdimensionalen Kettenregel

$\nabla_{\underline{x}}(g \circ \underline{\psi}(\underline{x})) = ((\nabla_{\underline{\xi}} g) \circ \underline{\psi}(\underline{x})) \cdot J_{\underline{\psi}}(\underline{x})$

und die Integraltransformation benötigt.

$$\int_{\mathbf{P}} g(\underline{\xi}) d\underline{\xi} = \int_{\underline{\psi}^{-1}(\mathbf{P})=\mathbf{Q}} g(\underline{\psi}(\underline{x}))|\det J_{\underline{\psi}}(\underline{x})| d\underline{x}$$

Mit der Jakobi-Matrix

$$J_{\underline{\psi}} = \begin{pmatrix} \frac{\partial \psi_1}{\partial x_1} & \frac{\partial \psi_1}{\partial x_2} \\ \frac{\partial \psi_2}{\partial x_1} & \frac{\partial \psi_2}{\partial x_2} \end{pmatrix}$$

Als neue Bilinearform ergibt sich

$$(2.4) \quad a^*(\hat{u}, \hat{v}) := \int_{\mathbf{P}} \nabla \hat{u} D^* (\nabla \hat{v})^T + \nabla \hat{u} \underline{b}^* \hat{v} + c^* \hat{u} \hat{v} d\underline{x}$$

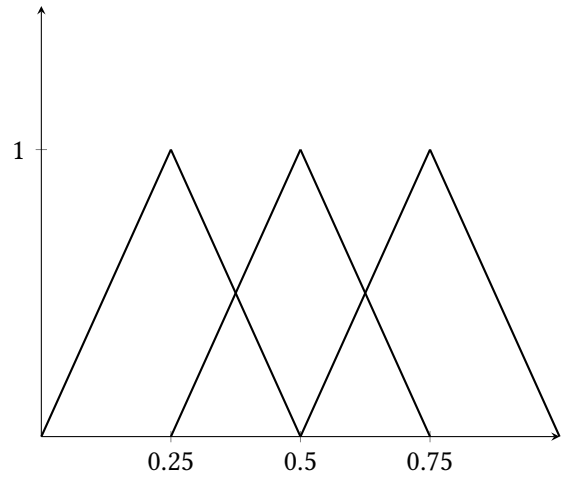
Die Funktionen mit Dach hängen hierbei von $\underline{\psi}(\underline{x})$ ab.

$$\begin{aligned} D^*(\underline{x}) &:= J_{\underline{\psi}}^{-1}(\underline{x})(D \circ \underline{\psi}(\underline{x})) J_{\underline{\psi}}^{-T}(\underline{x}) |\det J_{\underline{\psi}}(\underline{x})| \\ (2.5) \quad \underline{b}^*(\underline{x}) &:= J_{\underline{\psi}}^{-1}(\underline{x})(\underline{b} \circ \underline{\psi}(\underline{x})) |\det J_{\underline{\psi}}(\underline{x})| \\ c^*(\underline{x}) &:= (c \circ \underline{\psi}(\underline{x})) |\det J_{\underline{\psi}}(\underline{x})| \end{aligned}$$

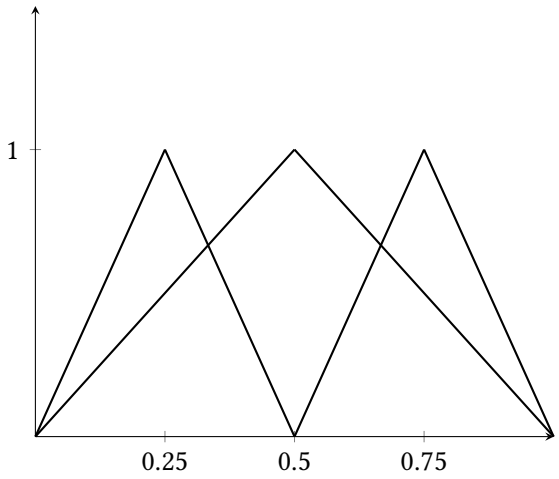
Auch die Berechnung der rechten Seite ändert sich mit der Transformation:

$f^*(\underline{x}) := (f \circ \underline{\psi}(\underline{x})) |\det J_{\underline{\psi}}(\underline{x})|$

2.2 Approximation und Erzeugendensystem



(a) Knotenbasis B_2



(b) Hierarchische Basis für Level 2

Um kontinuierliche Funktionen bearbeiten zu können, müssen diese zuerst diskretisiert werden. Hierzu wird eine Repräsentation auf einem diskreten Gitter gesucht, die der kontinuierlichen Funktion möglichst ähnlich ist. Wir bewegen uns auf dem Bereich [0,1] welcher in 2^l gleichgroße Teilstücke zerlegt wird. $h := 2^{-l}$ bezeichne den Abstand zwischen zwei Gitterpunkten. Nun lässt sich eine Funktion auf diesem Gitter als Linearkombination

$$F = \sum_{i=1}^{2^l-1} u_i \varphi_i$$

mit den Basisfunktionen φ_i und Koeffizienten u_i darstellen. Als Basisfunktion wird die Hütchenfunktion gewählt, welche wie folgt aussieht:

$$\varphi_i(x) = \begin{cases} 1 - |\frac{x - h \cdot i}{h}| & \text{wenn } (i-1) \cdot h \leq x \leq (i+1) \cdot h \\ 0 & \text{sonst} \end{cases} \tag{2.6}$$

Diese Funktionen sind einfach aufgebaut und gut geeignet um mit ihnen zu rechnen. Außerdem nehmen die Basisfunktionen eines Stützpunkts an allen anderen Stützpunkten den Wert 0 an. Deshalb reicht es zur Auswertung an einem Gitterpunkt den dazugehörigen Koeffizienten zu betrachten.

Eine alternative Methode zur Approximation ist die hierarchische Basis, welche auch bei den dünnen Gittern verwendet wird. Hierbei wird das Gitter in mehrere Level mit unterschiedlicher Maschenweite aufgeteilt, welche jeweils eigene Basisfunktionen besitzen. Die Gesamtfunktion wird durch Kombination aller Level dargestellt. In der hierarchischen Basis existiert für jeden Knotenpunkt nur eine Basisfunktion im gesamten Gitter, also haben feinere Level nicht mehr $2^l - 1$ Basisfunktionen sondern nur noch 2^{l-1}, die restlichen Punkte sind bereits auf gröberen Levels vorhanden. Dies hat zur Folge, dass die Darstellung einer Funktion auf dem Gitter eindeutig ist. Die hierarchische Basis

erlaubt es auch, bei Verfeinerung der Interpolation alle Koeffizienten der gröberen Level weiterhin zu verwenden und nur die des feinsten Levels hinzuzufügen.

Im Gegensatz zur hierarchischen Basis besitzt bei einem Erzeugendensystem jedes Level eine volle Knotenbasis, also alle $2^l - 1$ Gitterpunkte. Damit lässt sich auf Level l der gesamte Funktionsraum H_l darstellen. In der hierarchischen Basis ist dies nur durch Kombination mit gröberen Levels möglich. Die Darstellung einer Funktion durch Koeffizienten ist nun nicht mehr eindeutig, da die Anzahl der Freiheitsgrade die Anzahl der Stützpunkte übertrifft.

2.2.1 Das Prewavelet

Das Prewavelet ist eine alternative Basisfunktion, welche wie die Hütchenfunktion aus linearen Teilstücken besteht. Die Prewavelets verschiedener Levels sind zueinander L_2-orthogonal, eine

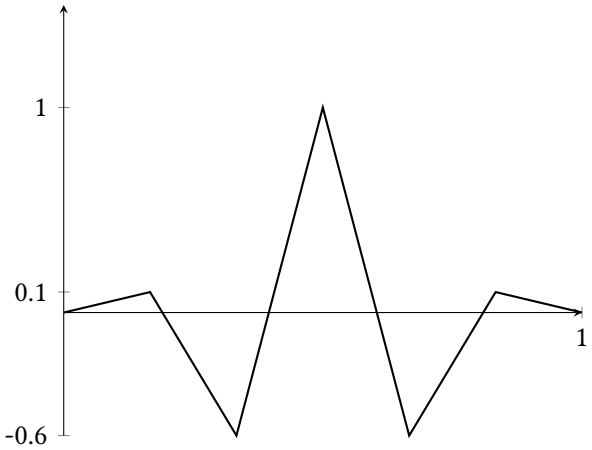


Abbildung 2.2: 1-dimensionale Prewavelet-Funktion

Eigenschaft die für den Algorithmus in dieser Arbeit wichtig ist. Allerdings ist das Rechnen mit Prewavelets unangenehm, da sie sich mit so vielen ihrer Nachbarn überlagern, und das Bilden von Differentialoperatoren komplex ist.

Hütchenfunktionen und Prewavelets spannen den selben Raum auf, im Erzeugendensystem lässt sich jedes Prewavelet der hierarchischen Basis als Kombination von Hütchenfunktionen des selben Levels repräsentieren:

$$\varphi_{l,i}^{pre}(x) = \begin{cases} \frac{9}{10}\varphi_{l,i} - \frac{3}{5}\varphi_{l,i+1} + \frac{1}{10}\varphi_{l,i-1} & \text{wenn } i = 1 \\ \varphi_{l,i} - \frac{3}{5}(\varphi_{l,i+1} + \varphi_{l,i-1}) + \frac{1}{10}(\varphi_{l,i+2} + \varphi_{l,i-2}) & \text{wenn } 3 \leq i \leq 2^l - 3 \\ \frac{9}{10}\varphi_{l,i} - \frac{3}{5}\varphi_{l,i+1} + \frac{1}{10}\varphi_{l,i-1} & \text{wenn } i = 2^l - 1 \end{cases}$$

Da die Darstellung einer Funktion keinen Einfluss auf das Ergebnis hat, ist es möglich in der Hütchenbasis zu rechnen, aber gleichzeitig die Semi-Orthogonalität der Prewavelets zu nutzen.

2.2.2 Mehrdimensionale Gebiete

Der Übergang vom eindimensionalen ins n-dimensionale geschieht indem das Tensorprodukt von n 1-D Gittern gebildet wird. Ein Level wird nun durch einen Multiindex $\underline{l}$ identifiziert. Es besitzt in Dimension d eine Maschenweite von $2^{-\underline{l}_i}$.

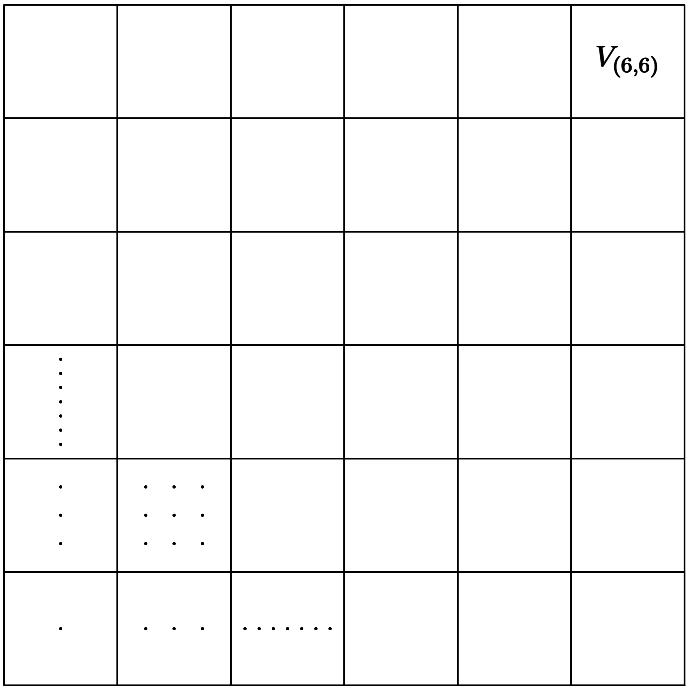


Abbildung 2.3: Alle vollen Gitter mit $max(\underline{l}) \leq 6$

Auch die Basisfunktionen im mehrdimensionalen werden mit Hilfe eines Tensorproduktes gebildet:

$$\varphi_{(\underline{l},\underline{i})}(\underline{x}) = \prod_k \varphi_{(l_k,i_k)}(x_k)$$

2.3 Dünne Gitter

Das Lösen des linearen Gleichungssystems $Au = f$ wird auf den bisher vorgestellten Gittern vor Allem im Mehrdimensionalen sehr teuer (Matrix A hat Größe O(N^{2d}) bei N Gitterpunkten pro Dimension). Deshalb wird in dieser Arbeit auf dünnen Gittern gerechnet. Die Idee hinter diesen Gittern ist, einen Großteil der Punkte des vollen Gitters, welcher nur einen geringen Beitrag zur Approximationsgenauigkeit liefert, zu vernachlässigen. Ein dünnes Gitter der Tiefe t ist wie folgt definiert:

$$V^{dünn} = \bigcup_{|\underline{l}| \leq t} V_{\underline{l}}^{voll}$$

mit

$$|\underline{l}| = \sum_{i=1}^{n} l_i$$

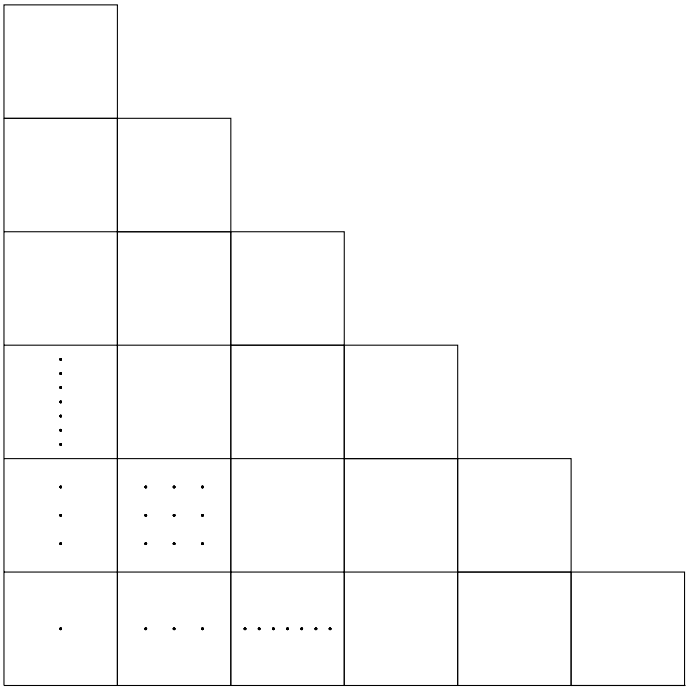


Abbildung 2.4: Anordnung der Gitter für $m = 6$ inklusive Gitterpunkte für Levels mit $l_1 + l_2 \leq 4$

Dies kann wie in Abbildung 2.4 dargestellt werden, alle Gitter über der Diagonalen sind für die Approximation nur von geringem Vorteil. Bild 2.5 zeigt ein solches dünnes Gitter. Durch diesen Ansatz verringert sich die Zahl der Gitterpunkte für ein Gitter von O(N^d) zu O($N(logN)^{d-1}$). Es lässt sich erkennen, dass für jede weitere Dimension die Anzahl der Gitterpunkte nur um einen Faktor von $logN$ steigt.

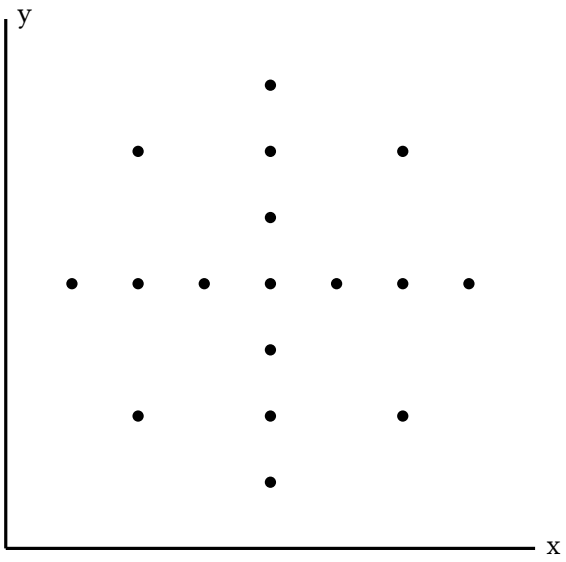


Abbildung 2.5: Vollständiges Dünnes Gitter

3 Das Verfahren

3.1 Operatoren

Um Werte zwischen den unterschiedlichen Levels des dünnen Gitters zu transportieren, werden im Folgenden die benötigten Operatoren definiert. Diese können in Matrixform dargestellt und auf einem Level in eine bestimmte Richtung angewendet werden. Dies geschieht, indem die Werte des Levels in Zeilen mit der selben Koordinate in r aufgeteilt werden, auf diese Zeilen wird der Operator durch einfache Matrixmultiplikation angewendet und anschließend die Zeilen wieder zusammengesetzt. Deshalb werden im Folgenden die 1-D Operatoren vorgestellt.

Der Prolongationsoperator $P_{l \leftarrow k}$ bewegt die Koeffizienten von einem gröberen Gitter k auf das nächstfeinere Level l, es gilt $l = k + 1$. Für die Hütchenfunktionen ergibt sich folgende Matrix als Prolongationsoperator:

$$P_{l \leftarrow k} = \begin{pmatrix} 0,5 & 0 & & & & \\ 1 & 0 & & & & \\ 0,5 & 0,5 & & & & \\ 0 & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & 0 \\ & & & 0,5 & 0,5 \\ & & & 0 & 1 \\ & & & 0 & 0,5 \end{pmatrix}$$

Diese Matrix besitzt $2^l - 1$ Zeilen und $2^k - 1$ Spalten.

Der umgekehrte Transport, also von feinem Level l auf gröberes Gitter k geschieht durch den Restriktionsoperator:

$$R_{k \leftarrow l} = \begin{pmatrix} 0,5 & 1 & 0,5 & & & & & & \\ 0 & 0 & 0,5 & 1 & 0,5 & & & & \\ & & & & \ddots & & & & \\ & & & & 0,5 & 1 & 0,5 & 0 & 0 \\ & & & & 0 & 0 & 0,5 & 1 & 0,5 \end{pmatrix}$$

Es gilt:

$$R_{k \leftarrow l} = (P_{l \leftarrow k})^T$$

Wenn die Level l und k nicht direkt nebeneinander liegen erfolgt der Transport durch Kombination der einzelnen Matrizen. Operatoren, die in unterschiedliche Richtungen wirken sind hierbei beliebig vertauschbar, nur die Reihenfolge der Operatoren in die selbe Richtung ist fix. Für den allgemeinen Prolongations- und Restriktionsoperator ist deshalb keine Richtungsangabe mehr vonnöten, da diese sich aus den Levels ergibt.

3.2 Die Steifigkeitsmatrix

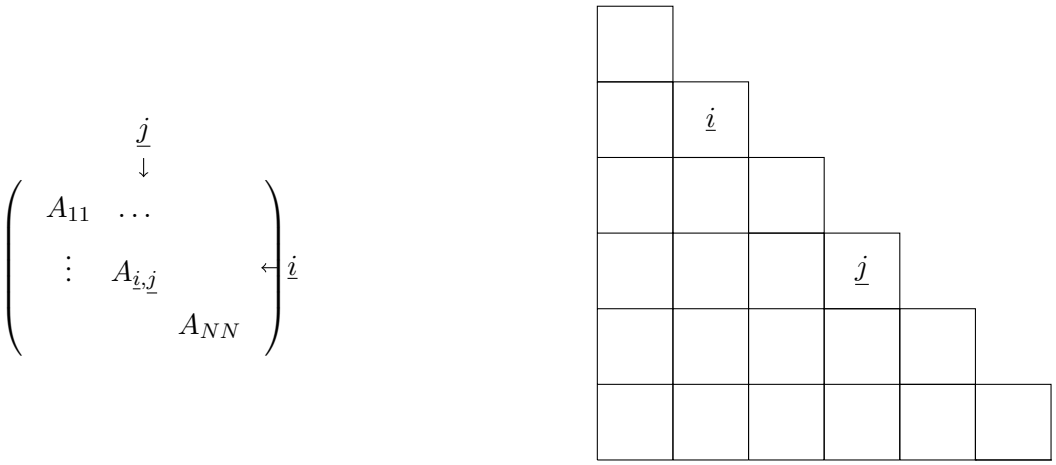


Abbildung 3.1: Block $A_{\underline{i},\underline{j}}$ entspricht dem Block der auf Gitter $\underline{j}$ angewendet wird und dessen Ergebnis auf Level $\underline{i}$ landet

Die Steifigkeitsmatrix A besteht aus Blöcken $A_{\underline{k},\underline{l}}$, welche den Kombinationen der Level $\underline{k}$ und $\underline{l}$ entsprechen (siehe Abbildung 3.2). Aufgrund der Symmetrie der Bilinearform ist auch die Steifigkeitsmatrix symmetrisch und es gilt $A_{\underline{k},\underline{l}} = A^T_{\underline{l},\underline{k}}$. Die Steifigkeitsmatrix eines Levels $\underline{l}$ wird abgekürzt als $A_{\underline{l},\underline{l}} = A_{\underline{l}}$. Anstatt $A_{\underline{l}}$ anzuwenden, kann auch ein Transport auf ein feineres Gitter $\underline{k}$ erfolgen und dort mit $A_{\underline{k}}$ multipliziert werden.

$$A_{\underline{l}} = R_{\underline{l}\leftarrow\underline{k}} A_{\underline{k}} P_{\underline{k}\leftarrow\underline{l}} \quad \text{mit } \underline{k} \text{ feiner } \underline{l}$$

Das genaue Aussehen der Steifigkeitsmatrix hängt von der partiellen Differentialgleichung, genauer gesagt von der dazugehörigen Bilinearform ab. In dieser Arbeit wird beispielhaft mit der Poissongleichung aus 3.1 im 2-D gerechnet.

(3.1) $\Delta u = f$

Aus der Poissongleichung ergibt sich als Bilinearform:

(3.2) $a(u,v) = \int \nabla u \nabla v d(x,y)$

Auf einem vollen Gitter mit Maschenweite h_x und h_y entsteht daraus der Neunpunktestern:

$$\frac{h_y}{6h_x}\begin{bmatrix}-1 & 2 & -1\\ -4 & 8 & -4\\ -1 & 2 & -1\end{bmatrix} + \frac{h_x}{6h_y}\begin{bmatrix}-1 & -4 & -1\\ 2 & 8 & 2\\ -1 & -4 & -1\end{bmatrix}$$

3.2.1 Behandlung krummberandeter Gebiete

Aus den Gleichungen 2.5 und 2.4 vorgestellt, ergibt sich der Differentialoperator auf einem krummberandeten Gebiet für die Poissongleichung als:

$$d^*(\hat{u},\hat{v}) := \int_{\Omega} \nabla\hat{u} D^* (\nabla\hat{v})^T d\underline{x} \tag{3.3}$$

$$D^*(\underline{x}) := J_{\underline{\psi}}^{-1}(\underline{x})(D \circ \underline{\psi}(\underline{x}))J_{\underline{\psi}}^{-T}(\underline{x})|\det J_{\underline{\psi}}(\underline{x})| \tag{3.4}$$

Die Matrix D ist in diesem Fall die Identitätsmatrix. Hiermit ergibt sich ein neuer Differentialoperator, dessen Koeffizienten von der Position auf dem Gitter abhängen. Da der Differentialoperator nun von nicht mehr nur vom Level, sondern auch von der Position an der er angewendet wird abhängt, muss sichergestellt werden, dass die Matrizen auf den einzelnen Levels konsistent bleiben. Um diesem Problem aus dem Weg zu gehen wird für diese Arbeit vorausgesetzt, dass die Steifigkeitsmatrix auf dem feinsten Gitter bekannt ist. Diese Matrix wird dann auf das Level, auf welchem sie benötigt wird restringiert. Somit ist die Konsistenz unter den Levels gegeben.

3.3 Grundidee

Wie schon in Kapitel 2.1 beschrieben wurde das Lösen der Differentialgleichung auf das lineare Gleichungssystem

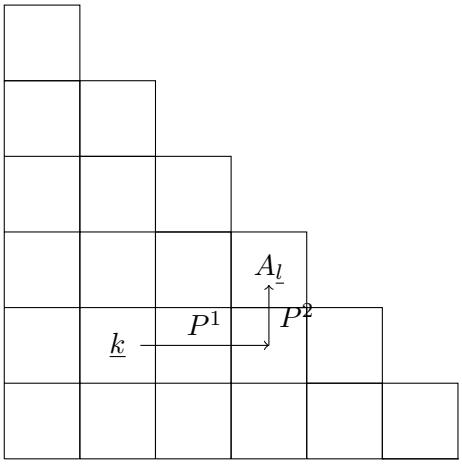
$$Au = f$$

zurückgeführt. Das Berechnen der rechten Seite der Gleichung stellt kein Problem dar, diese wird daher im Folgenden als bekannt angenommen. Der aufwändige Teil ist die Matrixmultiplikation, welche durch den hier vorgestellten Dünngitteralgorithmus gelöst werden soll.

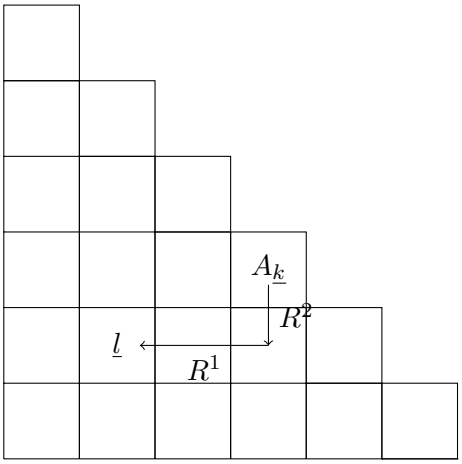
Um das Matrixprodukt auf einem Level $\underline{l}$ zu berechnen wird der Beitrag aller anderen Level benötigt. Auf Level $\underline{l}$ suchen wir:

$$\sum_{\forall \underline{k}} A_{\underline{k},\underline{l}} u_{\underline{k}} = \sum_{\forall \underline{k}} r_{\underline{l}\leftarrow\underline{k}}$$

Für das Ausgangslevel $\underline{k}$ gibt es 3 mögliche Positionen relativ zu $\underline{l}$



(a) $max(\underline{k}, \underline{l}) = \underline{l}$



(b) $max(\underline{k}, \underline{l}) = \underline{k}$

1. $max(\underline{k}, \underline{l}) = \underline{l}$

Wenn zu einem feineren Level Transportiert werden soll, ist es ausreichend zuerst auf das feinere Level zu prolongieren und anschließend dort die Steifigkeitsmatrix anzuwenden:

$$r_{\underline{l} \leftarrow \underline{k}} = A_{\underline{l}} P_{\underline{l} \leftarrow \underline{k}} u_{\underline{k}}$$

2. $max(\underline{k}, \underline{l}) = \underline{k}$

In diesem Fall kann mit der Steifigkeitsmatrix des feineren Levels multipliziert und anschließend auf $\underline{l}$ restringiert werden.

$$r_{\underline{l} \leftarrow \underline{k}} = R_{\underline{l} \leftarrow \underline{k}} A_{\underline{k}} u_{\underline{k}}$$

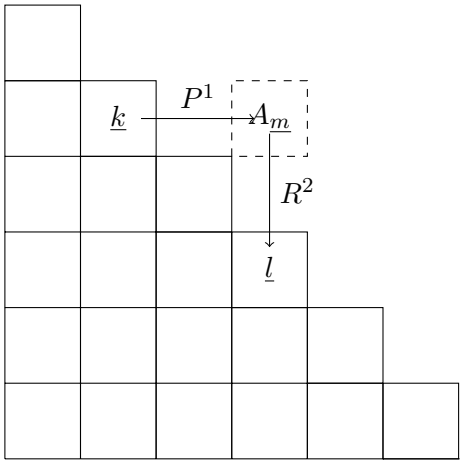
3. $\underline{k} \neq max(\underline{k}, \underline{l}) \neq \underline{l}$

Sei $\underline{m} = max(\underline{l}, \underline{k})$ und $\underline{\hat{m}} = min(\underline{l}, \underline{k})$. Falls sich die Anwendung der Steifigkeitsmatrix in die einzelnen Richtungen aufteilen lässt (im Allgemeinen nicht der Fall), dann ist der Transport über $\underline{\hat{m}}$ möglich (siehe Abbildung 3.3b, dies entspricht dem Algorithmus von Schwab und Todor). Sei o.B.d.A $l_1 < k_1$ und $l_2 > k_2$, dann gilt:

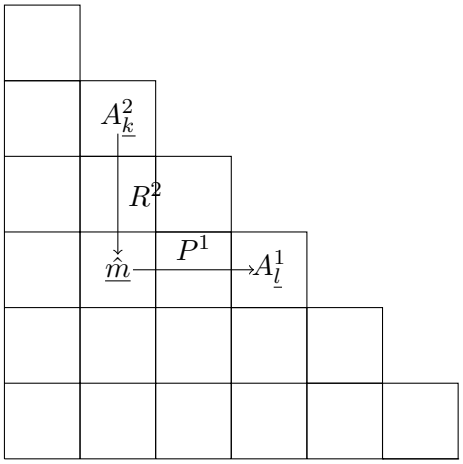
$$r_{\underline{l} \leftarrow \underline{k}} = A^2_{\underline{l}} P^2_{\underline{l} \leftarrow \underline{\hat{m}}} R^1_{\underline{\hat{m}} \leftarrow \underline{k}} A^1_{\underline{k}} u_{\underline{k}}$$

Da sich der Differentialoperator im Allgemeinen, und auch der hier gewählte Neunpunktestern für krummberandete Gebiete, nicht aufteilen lässt, muss der Transport über das feinste Gitter $\underline{m}$ erfolgen.

$$r_{\underline{l} \leftarrow \underline{k}} = R_{\underline{l} \leftarrow \underline{m}} A_{\underline{m}} P_{\underline{m} \leftarrow \underline{k}} u_{\underline{k}}$$



(a) $\underline{m}$ nicht im dünnen Gitter enthalten



(b) Transport mit Tensorprodukt-Operator

Für den Fall, dass $\underline{m}$ in dem dünnen Gitter enthalten ist, ist dieses Vorgehen auch unproblematisch. Wenn $\underline{m}$ aber nicht enthalten ist, dann ist die Multiplikation mit $A_{\underline{m}}$ zu teuer, im Extremfall ist $\underline{m}$ sogar das volle Gitter mit Größe $O(N^2)$.

Hier kommen nun die Prewavelets und das Erzeugendensystem wieder ins Spiel, die semi-Orthogonalität der Prewavelets erlaubt es, nach [RP] den Transport über Level, welche nicht im dünnen Gitter enthalten sind, zu ignorieren ohne die Güte der Lösung signifikant zu gefährden. Hieraus entsteht dann eine neue Bilinearform:

$$\tilde{a}(u_{\underline{l},j}, u_{\underline{k},i}) = \begin{cases} a(u_{\underline{l},j}, u_{\underline{k},i}) & \text{falls } max(\underline{l}, \underline{k}) \text{ in dünnem Gitter enthalten} \\ 0 & \text{sonst} \end{cases}$$

Diese neue Bilinearform entspricht einem Lumping der Steifigkeitsmatrix.

Da in dieser Arbeit aber mit den leichter zu behandelnden Hütchen als Basis gerechnet wird, muss die semi-Orthogonalität manuell sichergestellt werden. Dies ist möglich, da durch das Erzeugendensystem auf jedem Level der gesamte darunterliegende Raum aufgespannt wird, und auch die Prewavelets aus den Hütchen des Levels zusammengesetzt werden können.

Um die Orthogonalität auf Level $\underline{l}$ in Richtung r sicherzustellen wird der L_2-Projektionsoperator $\hat{L}^r_{\underline{k}\leftarrow\underline{l}}$ definiert:

$$\hat{L}^r_{\underline{k}\leftarrow\underline{l}} = (L^r_{\underline{k}})^{-1} R^r_{\underline{k}\leftarrow\underline{l}} L^r_{\underline{l}} \quad \text{mit } k_r = l_r - 1$$

Mit der L_2-Matrix:

$$L^r_{\underline{l}} = \frac{h_r}{6} \begin{pmatrix} 4 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ 0 & 1 & 4 & 1 & & & & \\ & & & \ddots & & & & \\ & & & 1 & 4 & 1 & 0 & \\ & & & 0 & 1 & 4 & 1 & \\ & & & 0 & 0 & 1 & 4 & \end{pmatrix}$$

Mit Gitterweite h_r in Richtung r auf Level $\underline{l}$.

Das Orthogonalisieren geschieht nun durch Anwenden der Projektionsmatrix und anschließendem Addieren des Ergebnisses zu $u_{\underline{k}}$:

$$u_{\underline{k}} \rightarrow u_{\underline{k}} + \hat{L}^r_{\underline{k}\leftarrow\underline{l}} u_{\underline{l}} \tag{3.5}$$

Nun muss noch sichergestellt werden, dass das Orthogonalisieren keinen Einfluss auf die Gesamtwerte hat, dies geschieht durch Gleichung 3.6

$$u_{\underline{l}} \rightarrow u_{\underline{l}} - P^r_{\underline{l}\leftarrow\underline{k}} \hat{L}^r_{\underline{k}\leftarrow\underline{l}} u_{\underline{l}} \tag{3.6}$$

Um die semi-Orthogonalität auch nach Änderung eines Wertes beizubehalten wird auf diesem, und allen darunterliegenden Level orthogonalisiert. Somit ist es möglich die gelumpte Steifigkeitsmatrix $\tilde{A}$ zu verwenden und den Transport über die nicht im dünnen Gitter enthaltenen Level zu vermeiden. Algorithmus 3.1 berechnet das Matrix-Vektor-Produkt auf Level $\underline{l}$, wenn das Produkt auf mehreren

Algorithmus 3.1 Theoretische Berechnung des Matrixproduktes auf Level $\underline{l}$

```
1:  function getProduct(Level l)
2:      product ← A_l u_l
3:      for all Level k ≠ l do
4:          m ← max(l, k)
5:          if m ∉ Levels then skip
6:          else if m = l then
7:              product ← product + A_l P_{l←k} u_k
8:          else if m = k then
9:              product ← product + R_{l←k} A_k u_k
10:         else
11:             product ← product + R_{l←m} A_m P_{m←k} u_k
12:         end if
13:     end for
14:     return product
15: end function
```

Levels gleichzeitig gesucht ist, lassen sich Zwischenergebnisse speichern, damit nicht mehrfach das selbe Produkt berechnet werden muss. Dieses Vorgehen wird in Algorithmus 4.3 genauer vorgestellt.

4 Gesamtalgorithmus und Implementierung in SG++

SG++ ist eine hauptsächlich in C++ geschriebene open-source Software Toolbox, welche speziell auf Dünngitterprobleme ausgerichtet ist [Pfl10]. Der Fokus liegt hierbei auf der effizienten Implementierung, weshalb alle für die Laufzeit kritischen Komponenten in C++ geschrieben sind. SG++ lässt sich auch in anderen Programmiersprachen wie Python, Matlab und Java benutzen.

4.1 Wichtige Module

SG++ besteht aus vielen Komponenten, die sich größtenteils einzeln aktivieren und deaktivieren lassen. Die für diese Arbeit notwendigen Module sind:

base

base definiert die grundlegende Funktionalität, und wird von allen anderen Modulen benötigt. Hierin wird auch die **Grid** Klasse definiert, von welcher die einzelnen dünnen Gitter abgeleitet werden. Auch die Klasse **DataVector** wird hier definiert. Ein DataVector enthält ein Array in welchem die Koeffizienten für ein dünnes Gitter oder Koordinaten auf einem Gitter gespeichert werden können. Eine weitere für diese Arbeit wichtige Klasse ist **OperationMatrix**, diese definiert eine Funktion für die Multiplikation einer Matrix mit den Koeffizienten einer Ansatzfunktion. Durch Ersetzen der **OperationMatrix.solve** Funktion, oder Implementieren einer alternativen Klasse, welche von **OperationMatrix** abgeleitet wird, und den in dieser Arbeit vorgestellten Algorithmus zur Matrixmultiplikation implementiert, kann das Lösungsverfahren der konjugierten Gradienten ohne sonstige Änderungen angewendet werden.

solver

Das Modul **solver** implementiert viele verschiedene Methoden für das Lösen von gewöhnlichen (Euler-Verfahren, Adams-Bashforth-Methode) und partiellen (Konjugierte Gradienten-Verfahren) Differentialgleichungen. Um mit der **ConjugateGradients** Klasse eine Gleichung zu lösen wird nur die rechte Seite der Gleichung sowie eine Implementierung der Multiplikation mit der Systemmatrix benötigt.

4.2 Implementierung in SG++

Die Werte $u_{\underline{l}}$

Die Koeffizienten eines Levels $\underline{l}$ werden als einzelner **DataVector** gespeichert. Dies vereinfacht den Transport zwischen den einzelnen Levels und die Anwendung der Steifigkeitsmatrix auf diesen.

Das Grid

Da keine Implementierung eines Gitters mit Erzeugendensystem in SG++ gefunden wurde, wird ein "normales"dünnes Gitter mit linearen Punktabständen als Ansatzpunkt gewählt. Wenn nun eine Gitterfunktion benötigt wird, kann durch einfache Transformation zwischen Erzeugendensystem und hierarchischer Basis gewechselt werden.

Die Transportoperatoren

Die Transportoperatoren wurden aus Performance-Gründen nicht als Matrixmultiplikation sondern explizit implementiert.

Der CG-Solver

Die bereits vorhandene Implementierung eines CG-Lösers wurde angepasst, indem die Matrixmultiplikation durch die in Algorithmus 4.3 vorgestellte Alternative ersetzt wurde.

Die Steifigkeitsmatrix

Die Steifigkeitsmatrix wird durch eine Galerkin-Vergröberung vom maximalen Level auf das jeweils benötigte Level angepasst.

4.3 Gesamtalgorithmus

Algorithmus 4.1 beschreibt in Pseudo-Code wie die Matrixmultiplikation zum Lösen von $Au = b$ verwendet werden kann.

In Algorithmus 4.2 wird die Orthogonalisierung aller Gitter im 2-D beschrieben.

Algorithmus 4.1 Lösen des linearen Gleichungssystems

```
1: function solveSLE(Matrix A, Vector b)
2:     MM ← MatrixMultiplication(A)
3:     CG ← ConjugateGradientSLE(MM)
4:     x ← CG.solve()
5:     return x
6: end function
```

Algorithmus 4.2 Orthogonalisieren aller Gitter im 2-D

```
1: procedure orthogonalize(Koeffizienten u)
2:     for y=max_level to 1 do
3:         for x=max_level_sum-y to 1 do
4:             if x > 1 then                                  // in Richtung 1 orthogonalisieren
5:                 temp ← R^1 L_{x,y} u_{x,y}                 // erster Teil der Projektion
6:                 temp ← solve(L_{x-1,y}, temp)              // auf dem groben Gitter mit L^{-1} multiplizieren
7:                 u_{x-1,y} ← u_{x-1,y} + temp
8:                 u_{x,y} ← u_{x,y} − P^1 temp
9:             end if
10:            if y > 1 then                                  // Richtung 2
11:                temp ← R^2 L_{x,y} u_{x,y}
12:                temp ← solve(L_{x,y-1}, temp)
13:                u_{x,y-1} ← u_{x,y-1} + temp
14:                u_{x,y} ← u_{x,y} − P^2 temp
15:            end if
16:        end for
17:    end for
18: end procedure
```

Algorithmus 4.3 beschreibt eine Möglichkeit das Berechnen des Produkts mithilfe von Zwischenergebnissen zu beschleunigen. Die Variablennamen geben hierbei die Position relativ zum zugehörigen Level an:

- $l \leftarrow$ links
- $r \leftarrow$ rechts
- $o \leftarrow$ oben
- $u \leftarrow$ unten

Und Kombinationen hieraus, dies wird in Bild 4.1 genauer dargestellt.

Algorithmus 4.3 Multiplikation von Matrix und Vector

```
1:  function MULT(Vector a)
2:      orthogonalize(a)
3:      for y=1 to max_level do
4:          for x=1 to max_level_sum-y do
5:              s_{x,y} ← A_{x,y} a_{x,y}
6:              if x > 1 then
7:                  l_{x,y} ← P^1(a_{x-1,y} + l_{x-1,y})
8:              end if
9:              if y > 1 then
10:                 u_{x,y} ← P^2(a_{x,y-1} + u_{x,y-1})
11:                 lu_{x,y} ← P^2(a_{x,y-1} + lu_{x,y-1} + l_{x,y-1})
12:             end if
13:         end for
14:     end for
15:     for x=max_level to 1 do
16:         for y=max_level_sum-x to 1 do
17:             if x > 1 then
18:                 r_{x-1,y} ← R^1(s_{x,y} + r_{x,y})
19:             end if
20:             if y > 1 then
21:                 lo_{x,y-1} ← R^2(s_{x,y} + A_{x,y} l_{x,y} + lo_{x,y})
22:                 ro_{x,y-1} ← R^2(r_{x,y} + ro_{x,y})
23:             end if
24:         end for
25:     end for
26:     for y=1 to max_level do
27:         for x=max_level_sum-y to 1 do
28:             if x > 1 then
29:                 ru_{x-1,y} ← R^1(A_{x,y} u_{x,y} + ru_{x,y} + s_{x,y})
30:             end if
31:             F_{x,y} ← A_{x,y}(l_{x,y} + lu_{x,y}) + lo_{x,y} + ro_{x,y} + ru_{x,y} + s_{x,y}
32:         end for
33:     end for
34:     return F
35: end function
```

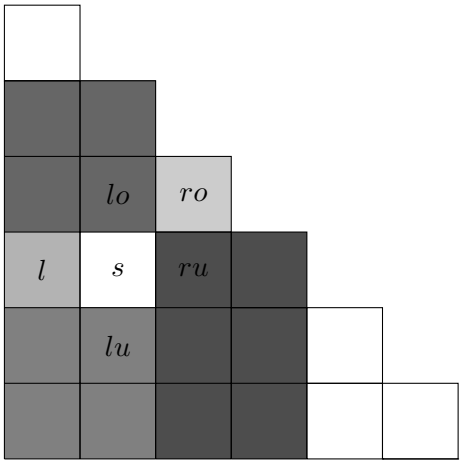


Abbildung 4.1: Zusammensetzung des Produktes auf einem Level

5 Ergebnisse

Zu Beginn der Vergleich zwischen dem in dieser Arbeit vorgestellten Algorithmus und dem Transport über das feinste Gitter in Bild 5.1. Die Laufzeit steigt deutlich schneller an, da der Transport auf das feinste Gitter, und die dortige Multiplikation, sehr teuer ist.

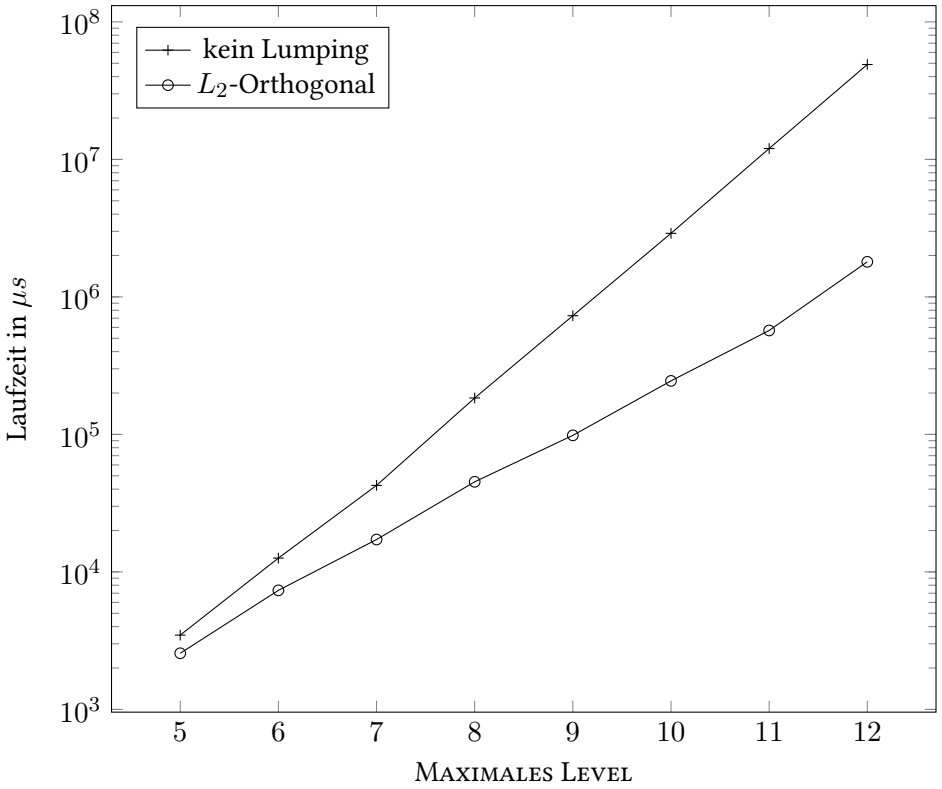


Abbildung 5.1: Laufzeit einer Matrixmultiplikation mit und ohne Lumping

Hier werden nun 2 Alternativen zum Orthogonalisieren mithilfe der L_2-Projektion gegeben:

- Lumping der L_2-Matrix, damit die L_2-Projektion schneller zu berechnen ist
- Einfaches Hierarchisieren

Damit ist zwar keine Orthogonalität mehr garantiert, aber der Fehler verhält sich auch nach Lumping der Steifigkeitsmatrix in der selben Größenordnung wie ohne Lumping (Abbildung 5.2), unabhängig davon, ob die normale L_2-Projektion oder eine der Alternativen angewendet werden.

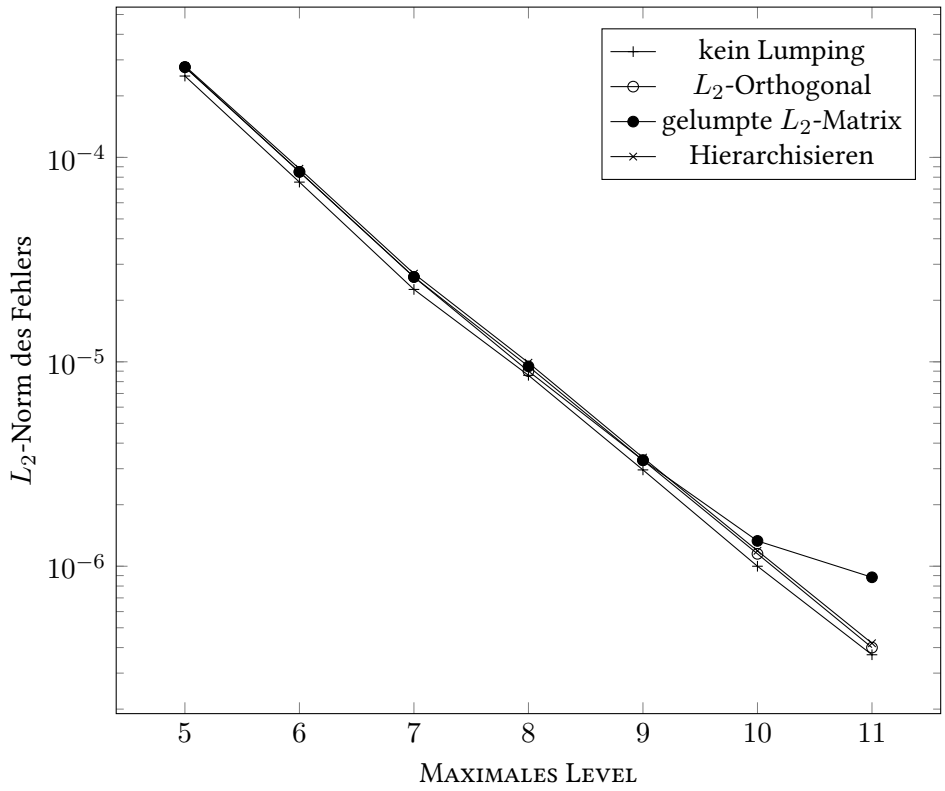


Abbildung 5.2: L_2-Norm des Fehlers auf dem krummen Gebiet mit und ohne Lumping, die 3 Methoden des Lumping liefern sehr ähnliche Ergebnisse

Als Beispiel für ein Gebiet mit krummem Rand wird gewählt:

$$\xi_1 = x_1 + \frac{x_2^2}{10}$$
$$\xi_2 = x_2 + 2x_1(1 - x_1)$$

Auch hier verhalten sich die Fehler ähnlich (Bild 5.3).

Abschließend noch ein Vergleich der Laufzeit für das orthogonalisieren, beziehungsweise äquivalente Operationen in Abbildung 5.4. Die L_2-Projektion schneidet aufgrund der Matrixinversen am schlechtesten ab.

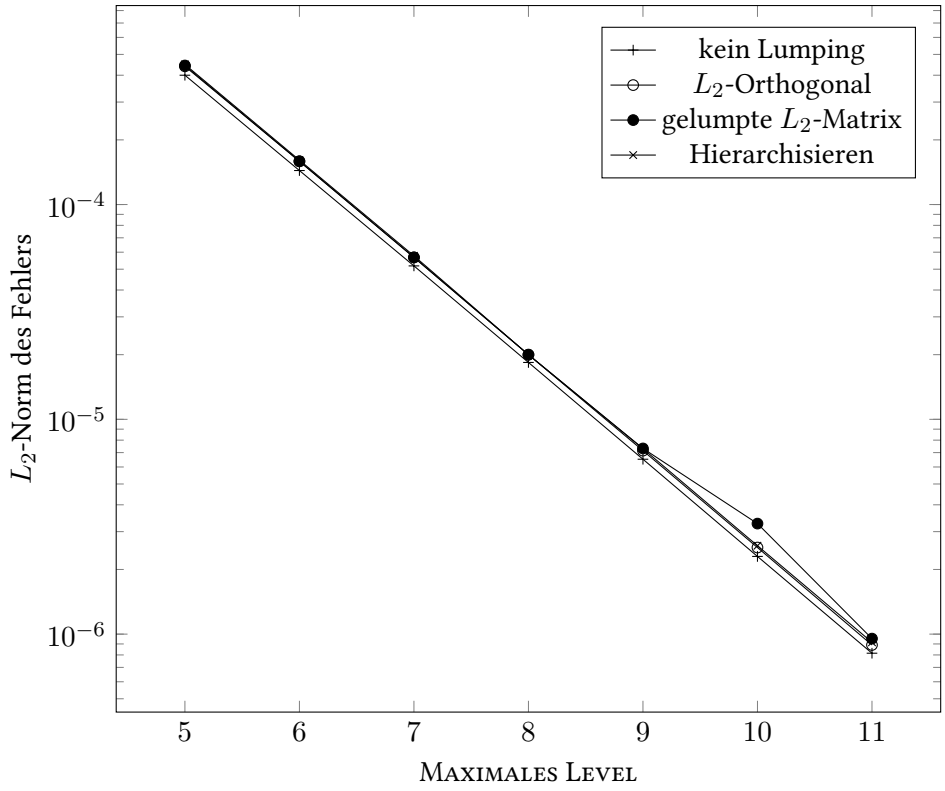


Abbildung 5.3: L_2-Norm des Fehlers auf dem krummen Gebiet mit und ohne Lumping, die 3 Methoden des Lumping liefern sehr ähnliche Ergebnisse

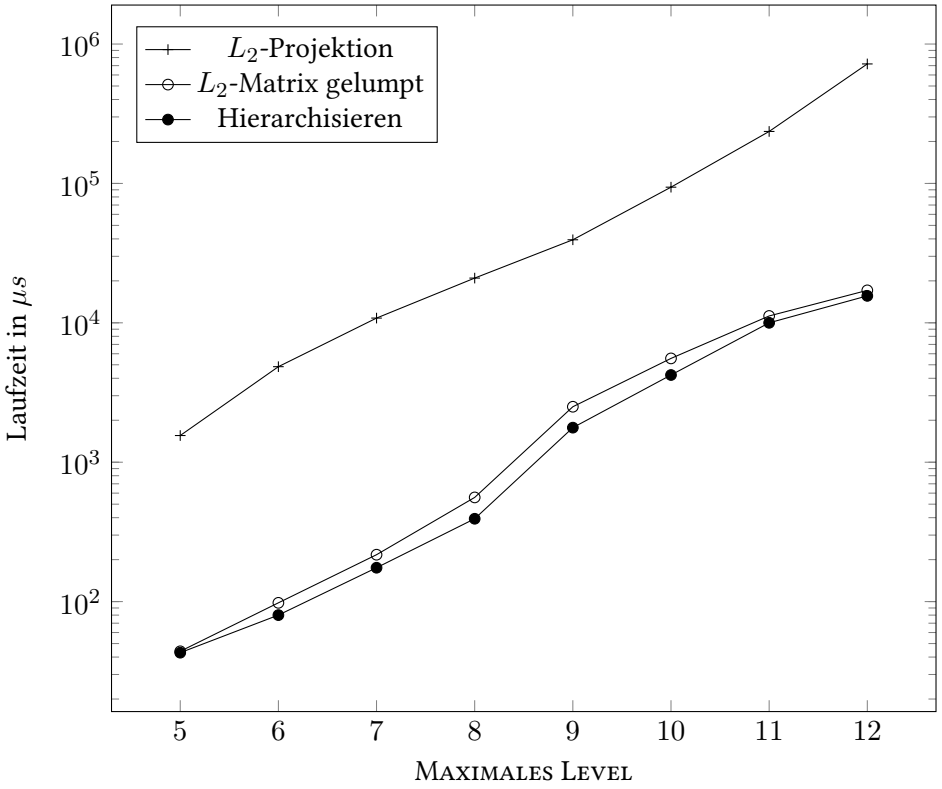


Abbildung 5.4: Laufzeit der Örthogonalisierung"

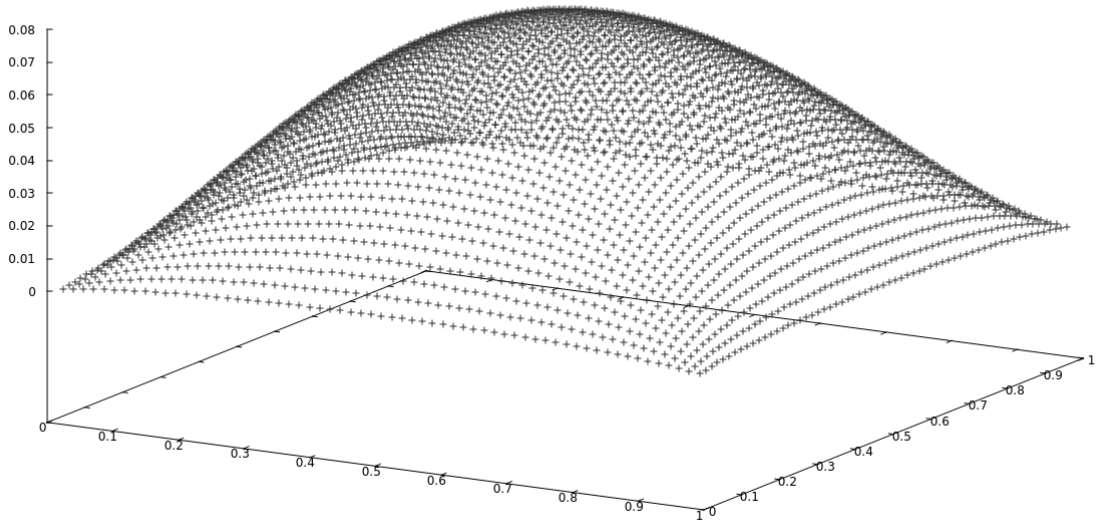


Abbildung 5.5: Approximation der Lösung der Poisson-Gleichung für Au=1 auf dem Einheitsquadrat

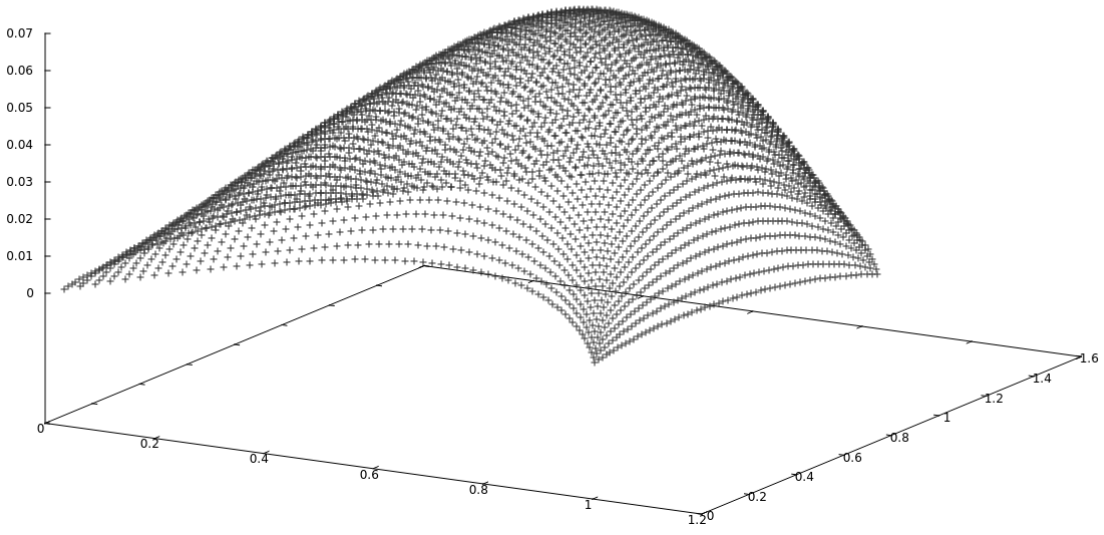


Abbildung 5.6: Auf dem transformierten Gebiet

6 Ausblick

In dieser Arbeit wurde ein Algorithmus zur effizienten Multiplikation einer Steifigkeitsmatrix mit einem Vektor vorgestellt, welcher dazu verwendet werden kann, partielle Differentialgleichungen zu lösen, auch wenn deren Operator keine Tensorproduktform hat.

Dies wurde nur im zweidimensionalen vorgestellt, lässt sich jedoch einfach auch in höhere Dimensionen erweitern, im 3-D würde aus dem 9-Punktestern dann ein 27-Punktestern.

Auch der Differentialoperator kann ohne Weiteres angepasst werden.

Literaturverzeichnis

[Dor97] T. Dornseifer. *Diskretisierung allgemeiner elliptischer Differentialgleichungen in krummlinigen Koordinatensystemen auf dünnen Gittern.* na, 1997. (Zitiert auf Seite 6)

[Hac96] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen.* Teubner Verlag, 1996. (Zitiert auf Seite 5)

[Pfl10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems.* Verlag Dr. Hut, München, 2010. URL http://www5.in.tum.de/pub/pflueger10spatially.pdf. (Zitiert auf Seite 18)

[RP] H. R., C. Pflaum. A Sparse Grid Discretization with Variable Coefficient in High Dimensions. *SIAM Journal on Numerical Analysis.* (Zitiert auf den Seiten 4 und 16)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift