Institut für Parallele und Verteilte Systeme
Abteilung Simulation großer Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3276

Integration von Fluidsimulationen in die Parallel Multilevel Partition of Unity Methode

Christian Dittrich

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Marc Alexander Schweitzer
Betreuer:	Dr. rer. nat. Stefan Zimmer
begonnen am:	12. Dezember 2011
beendet am:	12. Juni 2012
CR-Klassifikation:	F.2.1, G.1.3, G.1.8, J.2

Kurzfassung

In dieser Diplomarbeit werden zwei Strategien für die Implementierung der Partition of Unity Methode, insbesondere zum Lösen von Sattelpunktproblemen, beschrieben und prototypisch durchgeführt. Die entstandenen Prototypen dienen als Basis für experimentelle Umsetzungen von alternativen Algorithmen und Optimierungen. Bei der Implementierung der Prototypen standen daher gute Lesbarkeit, Simplizität, Verständlichkeit und Erweiterbarkeit im Vordergrund und weniger eine möglichst geringe Rechenzeit. Es wird gezeigt, wie dieses Ziel in einem engen Zeitrahmen mit einem iterativen Prozess erreicht werden kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Die Beschreibung von physikalischen Sachverhalten und Problemen erfolgt in der Regel mittels Differentialgleichungen und Differentialgleichungssysteme. Für diese lässt sich im Allgemeinen keine analytische Lösung finden. Numerische Verfahren können hier Abhilfe schaffen, indem mit ihnen eine Lösung näherungsweise berechnet wird. Solche numerischen Verfahren sind beispielsweise die Finite Differenzen Methode und die Finite Elemente Methode. Als Grundlage für klassische Finite Differenzen und Elemente dient immer ein Gitter. Das verwendete Gitter ist maßgebend für die Güte der Lösung. Deshalb wird bei Lösungen mit Finiten Elemente ein erheblicher Teil des Aufwands in die Erzeugung eines guten Gitters aufgewendet.

Die Partition of Unity Methode ist eine erweiterte Finite Elemente Methode, die ohne Gitter funktioniert, sodass sich der Aufwand für die Gittererzeugung einsparen lässt. Voraussetzung dafür ist, dass eine gleiche Genauigkeit bei der Lösung erreicht wird, wie bei gitterbasierten Finiten Elementen. Ein weiterer Vorteil der Partition of Unity Methode ist, dass sich Wissen über das lokale Verhalten der Lösung sehr einfach in den Ansatzraum integrieren lässt und damit die Lösung signifikant verbessert werden kann.

Ziel der Diplomarbeit war es, die Lösung von Sattelpunktproblemen mit der Partition of Unity Methode prototypisch zu implementieren. In Kapitel 2 werden dafür die Grundlagen erläutert. Es wird dabei davon ausgegangen, dass der Leser mit der klassischen Finite Elemente Methode und Differentialgleichungen im Allgemeinen vertraut ist. Im Rahmen dieser Diplomarbeit sind zwei Prototypen für die Implementierung der Partition of Unity Methode entwickelt und in Kapitel 3 dokumentiert. Die Prototypen werden nach Verständlichkeit, Leistungsverhalten und Erweiterbarkeit verglichen. Abschließend folgt ein Fazit gezogen und mögliche Weiterentwicklungen aufgezeigt.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen eingeführt, auf denen die beiden implementierten Prototypen beruhen. Dabei werden alle wichtigen Begriffe erläutert, die in Kapitel 3 verwendet werden.

2.1 Variationsformulierung und Galerkin-Verfahren

Gegeben sind ein Gebiet Ω und ein Hilbertraum S von Funktionen auf diesem Gebiet. Sei S_h ein endlichdimensionaler Teilraum von S, dann hat die Variationsaufgabe

$$J(v) = \frac{1}{2} a(v,v) - l(v) \to \min v \in S_h \tag{2.1}$$

für eine symmetrisch, positiv definite Bilinearform $a(\cdot,\cdot)$ und eine Linearform $l(\cdot)$ in S_h ein Minimum bei $u_h \in S_h$, wenn

$$a(u_h, v) = l(v) \tag{2.2}$$

für alle $v \in S_h$ gilt (siehe [Brao7]). Sei $\{\phi_1, \ldots, \phi_n\}$ eine Basis von S_h und $u_h = \sum_{i=1}^{n} u_i \phi_i$, dann ist 2.2 äquivalent zu

$$\sum_{j=1}^{n} u_j a\left(\phi_j, \phi_i\right) = l(\phi_i) \text{ für } i = 1, \ldots, n \tag{2.3}$$

Durch Definition von

$$\begin{aligned} A_{i,j} &= a\left(\phi_j, \phi_i\right) \\ b_i &= l(\phi_i) \end{aligned} \tag{2.4}$$

ergibt sich ein lineares Gleichungssystem in Matrix-Vektor-Schreibweise $Au = b$. Wie eine Basis für einen Raum S_h konstruiert werden kann, wird in Abschnitt 2.2 gezeigt.

2.1.1 Approximationsproblem

Sei $f \in S = L^2(\Omega)$ und $u \in S_h$, dann definiert

$$u = f \quad \text{auf } \Omega \tag{2.5}$$

das Approximationsproblem in der L^2-Norm. Die dafür verwendete Bilinearform und Linearform sehen wie folgt aus:

$$\begin{aligned} a(\phi_j, \phi_i) &= \int_\Omega \phi_j \phi_i \\ l(\phi_i) &= \int_\Omega f \phi_i \end{aligned} \tag{2.6}$$

2.1.2 Poisson-Problem

Die Poisson-Gleichung wird auch Potentialgleichung genannt und beschreibt beispielsweise das elektrostatische Potential und das Gravitationspotential.

$$\begin{aligned} \Delta u &= -f \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{2.7}$$

Für homogene Randwerte $g = 0$ sehen die dazugehörige Bilinearform und Linearform sehen wie folgt aus:

$$\begin{aligned} a(\phi_j, \phi_i) &= \int_\Omega \nabla\phi_j \nabla\phi_i \\ l(\phi_i) &= \int_\Omega f \phi_i \end{aligned} \tag{2.8}$$

Eine Bilinearform und Linearform für inhomogene Randwerte wird in Abschnitt 2.4 definiert.

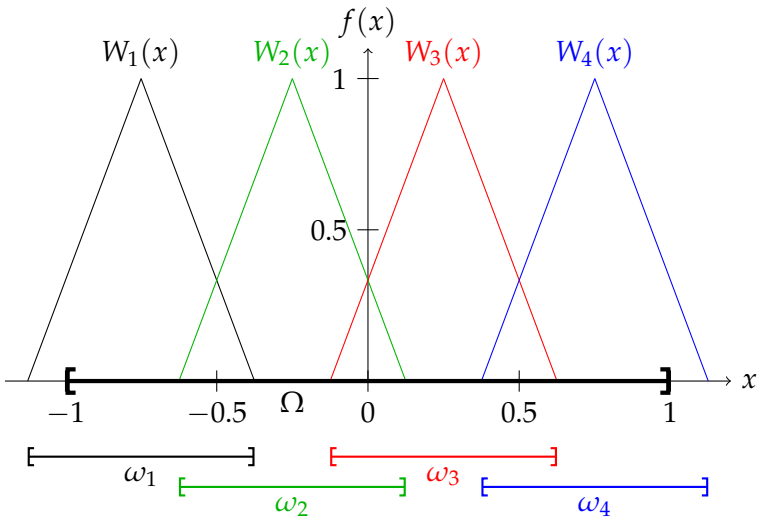

Abbildung 2.1: Gewichtsfunktionen auf dem Gebiet $\Omega = [-1, 1]$ überdeckt durch ein Cover aus vier Patches.

2.2 Partition of Unity Methode

Die Partition of Unity Methode (PUM) [MB96] ist eine erweiterte Finite Element Methode (FEM), die ohne Gitter auskommt. Bei Finite Element Methoden stellt die Verknüpfung der Gitterzellen die globale Stetigkeit sicher. Da das Gitter bei der Partition of Unity Methode wegfällt, muss hier ein anderer Ansatz gefunden werden. Dazu wird eine Menge von Funktionen, die sogenannten PUM-Funktionen, konstruiert, die als eine Art Kleber zwischen den lokalen Approximationsräumen dient. Diese lassen sich mittels Shepard-Konstruktion konstruieren.

2.2.1 Shepard-Konstruktion

Als Basis für die Konstruktion der PUM-Funktionen wird eine Methode zur Interpolation zwischen Datenpunkten gewählt. Die Scattered Data Approximation findet mittels Gewichtung der zu interpolierenden Werte $u_i = u(x_i)$ mit $i = 1, \ldots, n$ zu jedem Punkt x einen korrespondieren Wert $u(x)$.

$$u(x) = \sum_{i=1}^{n} u_i \varphi_i(x) \tag{2.9}$$

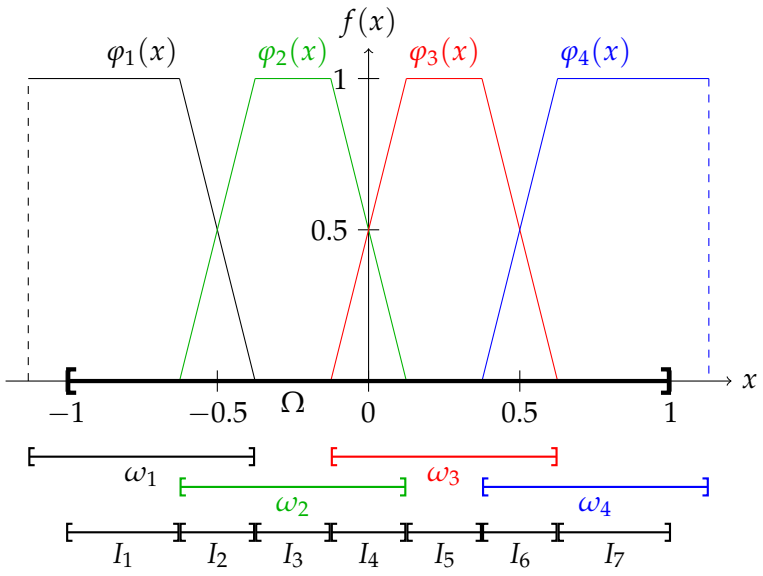

Abbildung 2.2: Aus den Gewichtsfunktionen resultierende PUM-Funktionen und Integrationszellen.

Shepard definiert die Funktionen φ_i in [She68] mittels Inverse Distance Weighting. Dabei hat der Wert u_i an einem Punkt x_i umso mehr Einfluss auf den interpolierten Wert $u(x)$, je näher x an x_i liegt:

$$\varphi_i(x) = \frac{W_i(x)}{\sum_{j=1}^{n} W_j(x)} \text{ mit } W_i(x) = \frac{1}{\|x - x_i\|^\beta} \tag{2.10}$$

Diese Funktionen haben die Eigenschaft, dass sie aufsummiert immer Eins ergeben. Daher der Name Partition of Unity (Partition der Eins).

$$\sum_{i=1}^{n} \varphi_i = \sum_{i=1}^{n} \frac{W_i(x)}{\sum_{j=1}^{n} W_j(x)} = \frac{\sum_{i=1}^{n} W_i(x)}{\sum_{j=1}^{n} W_j(x)} \equiv 1 \tag{2.11}$$

Der Nachteil dieser Gewichtsfunktionen W_i liegt darin, dass diese einen globalen Träger haben. Zur Auswertung einer Funktion φ_i an einer Stelle x müsste man alle n Gewichtsfunktionen W_i auswerten. Ein weiterer Nebeneffekt der globalen Träger wäre, dass die Matrix in einem Galerkin-Verfahren dicht besetzt wäre, wodurch die Matrix $O(n^2)$ Speicherplatz benötigen würde.

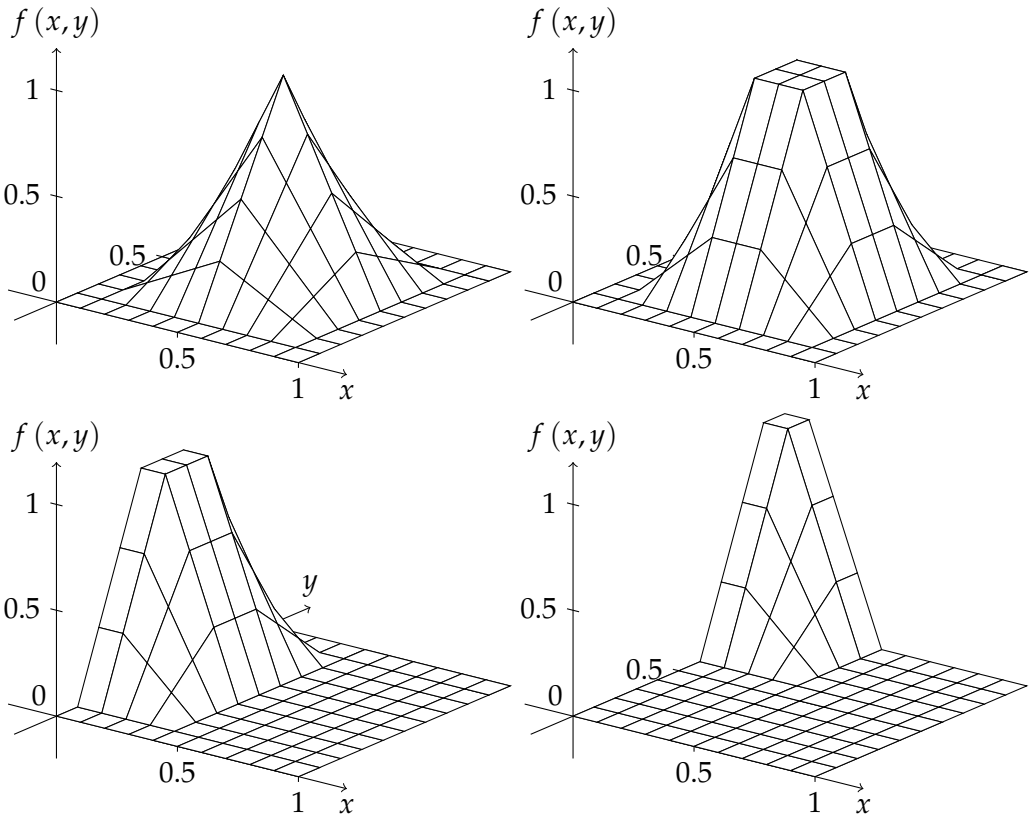

Abbildung 2.3: von oben links nach unten rechts: Zweidimensionale Gewichtsfunktion auf dem Patch $\omega = [0.1, 0.9] \times [0.1, 0.9]$; zweidimensionale PUM-Funktion im Inneren eines Gebiets ($\omega \cap \partial\Omega = \emptyset$), am Rand eines Gebiets ($\omega \cap \partial\Omega \neq \emptyset$) und in der Ecke eines Gebiets.

Um diese Probleme zu umgehen, verwendet man für die Partition of Unity Gewichtsfunktionen mit lokalem Träger. Solche Gewichtsfunktionen lassen sich beispielsweise durch Splines konstruieren. Die einfachsten dieser Splines sind lineare Splines. Die sich daraus ergebende Gewichtsfunktion sieht wie folgt aus:

$$W_i(x) = \begin{cases} 2\frac{x-a_i}{b_i-a_i} & a_i < x \leq \frac{a_i+b_i}{2} \\ 2\frac{b_i-x}{b_i-a_i} & \frac{a_i+b_i}{2} < x < b_i \\ 0 & x \notin \omega_i \end{cases} \tag{2.12}$$

wobei $\omega_i = [a_i, b_i]$ der Träger der Funktion ist. Diese Träger entsprechen bei der Partition of Unity Methode den Patches des Covers. Durch die lokalen Träger ist es nun nicht mehr nötig für die Auswertung einer PUM-Funktion φ_i an der Stelle x alle Gewichtsfunktionen

auszuwerten. Es müssen lediglich alle ausgewertet werden, die an der Stelle x nicht null sind.

$$\varphi_i(x) = \frac{W_i(x)}{\sum_{\omega_i \cap \omega_j \neq \emptyset} W_j(x)} \tag{2.13}$$

Wie sich aus den Gewichtsfunktionen 2.12 die PUM-Funktionen ergeben, zeigten die Abbildungen 2.1 und 2.2 für ein Cover von vier Patches über dem Gebiet $\Omega = [-1, 1]$. Für zweidimensionale Gewichtsfunktionen wird einfach das Tensorprodukt aus zwei eindimensionalen Gewichtsfunktionen gebildet:

$$W_k(x, y) = W_i(x) W_j(y) \tag{2.14}$$

Die Konstruktionsvorschrift für die PUM-Funktionen bleibt im Zweidimensionalen erhalten (siehe Abbildung 2.3):

$$\varphi_i(x, y) = \frac{W_i(x, y)}{\sum_{\omega_i \cap \omega_j \neq \emptyset} W_j(x, y)} \tag{2.15}$$

2.2.2 Lokale Approximationsräume

Die in Abschnitt 2.2.1 vorgestellten PUM-Funktionen sind noch nicht ausreichend, um einen Raum S_h zu konstruieren, da nur konstante Funktionen für die lokale Approximation verwendet werden. Die Form der PUM-Funktionen ermöglicht es lokale Approximationsräume $V_i^{p_i} = \{\psi_0, \ldots, \psi_{p_i}\}$ aus lokalen Ansatzfunktionen ψ_j stetig miteinander zu verknüpfen. Diese lokalen Approximationsräume können für jeden Patch beliebig festgelegt werden, die globale Stetigkeit wird allein durch die PUM-Funktionen sichergestellt. Die lokalen Approximationsfunktionen müssen nur ausreichend glatt sein. Dadurch wird die lokale Approximationseigenschaft hergestellt, die für ein Galerkin-Verfahren benötigt wird.

Der daraus resultierende globale Approximationsraum $S_h = \{\phi_{1,0}, \ldots, \phi_{n,m}\} = \{\varphi_1\psi_0, \ldots, \varphi_n\psi_m\}$ lässt sich durch die Unabhängigkeit der lokalen Approximationsräume voneinander sogar mit Informationen über das lokale Verhalten der analytischen Lösung anreichern, wodurch der Approximationsfehler signifikant verringert werden kann. Dafür werden Funktionen, die das lokale Verhalten der analytischen Lösung beschreiben, in den lokalen Approximationsraum des betroffenen Patches eingeführt. Diese Funktionen werden auch als Enrichments bezeichnet. Die Funktionen $\phi_{i,j} = \varphi_i\psi_j$ werden als Shape-Funktionen bezeichnet.

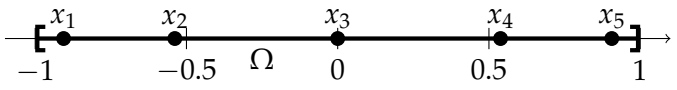

Abbildung 2.4: Auswertungspunkte x_i für die 5-Punkt-Gauß-Quadratur.

i	x_i	α_i
1	$-\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$\frac{322-13\sqrt{70}}{900}$
2	$-\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{322+13\sqrt{70}}{900}$
3	0	$\frac{128}{225}$
4	$\frac{1}{3}\sqrt{5-2\sqrt{\frac{10}{7}}}$	$\frac{322+13\sqrt{70}}{900}$
5	$\frac{1}{3}\sqrt{5+2\sqrt{\frac{10}{7}}}$	$\frac{322-13\sqrt{70}}{900}$

Tabelle 2.1: Auswertungspunkte x_i und Gewichtungen α_i für die 5-Punkt-Gauß-Quadratur.

In den hier angeführten Beispielen wird in der Regel auf die lokale Ansatzfunktion verzichtet und der Einfachheit halber nur mit den PUM-Funktionen φ_i gearbeitet. Dies ist ohne Probleme möglich, da die lokalen Ansatzfunktionen beliebig glatt sind und keine Implikationen für die Berechnung der Bilinearformen respektive der Integrale haben. Wenn beispielsweise im Folgenden die Rede von einem Integral $\int_\Omega \varphi_i \varphi_j$ ist, dann ist damit ein Integral der Form $\int_\Omega \phi_{i,k}\phi_{j,l} = \int_\Omega \varphi_i \psi_k \varphi_j \psi_l$ gemeint, wenn nicht anders angegeben.

2.3 Numerische Integration

Die Auswertung der Bilinearform $a(\cdot,\cdot)$ aus Abschnitt 2.1 beinhaltet die Auswertung von Integralen der Form $\int_\Omega \phi_{i,k}\phi_{j,l}$. Diese Integrale sind für die unter Abschnitt 2.2 vorgestellten Funktionen $\phi_{i,k}$ zwar analytisch lösbar, was aber für eine programmatische Umsetzung nicht in Frage kommt. Insbesondere die analytische Integration von Enrichments ist im Allgemeinen nicht gegeben.

Sind Integrale nicht analytisch lösbar, so lassen sie sich doch zumindest näherungsweise numerisch berechnen. Dazu wird die Funktion an vorgegeben Stützstellen x_i ausgewertet und eine gewichtete Summe gebildet:

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=1}^{n} \alpha_i f(x_i) \tag{2.16}$$

Die in den Protoypen verwendete Gauss-Quadratur kann Polynome bis zu einem Grad $2n+1$ exakt berechnen, wobei n die Anzahl der Stützstellen ist. Abbildung 2.4 und Tabelle 2.1 zeigen die Gewichte und Stützstellen für die verwendete 5-Punkt-Gauss-Quadratur. Damit die Gauss-Quadratur bis zum Grad $2n+1$ exakt ist, darf die Funktion im Integrationsintervall keine Knicke haben. Daher wird das Integrationsintervall für PUM-Funktionen in Integrationszellen unterteilt, auf denen sie keine Knicke haben. Diese Knicke sind für PUM-Funktionen durch die Nachbarschaft der Patches bestimmt.

Durch die Lokalität der Träger der PUM-Funktionen ist es nicht nötig jedes Integral auf allen Integrationszellen auszuwerten. Dazu soll das Beispiel aus Abbildung 2.2 dienen. Zu sehen ist, dass das Integral $\int_\Omega \varphi_1\varphi_1$ nur auf den Integrationszellen I_1 und I_2 zu nicht null ausgewertet wird:

$$\begin{aligned}
\int_\Omega \varphi_1\varphi_1 &= \int_{I_1\cup I_2\cup I_3\cup I_4\cup I_5\cup I_6\cup I_7} \varphi_1\varphi_1 \\
&= \int_{I_1} \varphi_1\varphi_1 + \int_{I_2} \varphi_1\varphi_1 + \int_{I_3} \varphi_1\varphi_1 + \int_{I_4} \varphi_1\varphi_1 + \int_{I_5} \varphi_1\varphi_1 + \int_{I_6} \varphi_1\varphi_1 + \int_{I_7} \varphi_1\varphi_1 \\
&= \int_{I_1} \varphi_1\varphi_1 + \int_{I_2} \varphi_1\varphi_1 \\
&\text{mit } \int_{I_3} \varphi_1\varphi_1 = \int_{I_4} \varphi_1\varphi_1 = \int_{I_5} \varphi_1\varphi_1 = \int_{I_6} \varphi_1\varphi_1 = \int_{I_7} \varphi_1\varphi_1 = 0
\end{aligned} \tag{2.17}$$

Diese Vereinfachung von Integralen über alle Integrationszellen auf wenige Integrationszellen führt dazu, dass der Aufwand für die Integration erheblich gesenkt wird:

$$
\begin{aligned}
&\begin{pmatrix}
\int_\Omega \varphi_1\varphi_1 & \int_\Omega \varphi_2\varphi_1 & \int_\Omega \varphi_3\varphi_1 & \int_\Omega \varphi_4\varphi_1 \\
\int_\Omega \varphi_1\varphi_2 & \int_\Omega \varphi_2\varphi_2 & \int_\Omega \varphi_3\varphi_2 & \int_\Omega \varphi_4\varphi_2 \\
\int_\Omega \varphi_1\varphi_3 & \int_\Omega \varphi_2\varphi_3 & \int_\Omega \varphi_3\varphi_3 & \int_\Omega \varphi_4\varphi_3 \\
\int_\Omega \varphi_1\varphi_4 & \int_\Omega \varphi_2\varphi_4 & \int_\Omega \varphi_3\varphi_4 & \int_\Omega \varphi_4\varphi_4
\end{pmatrix} \\
&= \begin{pmatrix}
\int_{I_1} \varphi_1\varphi_1 + \int_{I_2} \varphi_1\varphi_1 & \int_{I_2} \varphi_2\varphi_1 & 0 & 0 \\
\int_{I_2} \varphi_1\varphi_2 & \int_{I_2} \varphi_2\varphi_2 + \int_{I_3} \varphi_2\varphi_2 + \int_{I_4} \varphi_2\varphi_2 & \int_{I_4} \varphi_3\varphi_2 & 0 \\
0 & \int_{I_4} \varphi_2\varphi_3 & \int_{I_4} \varphi_3\varphi_3 + \int_{I_5} \varphi_3\varphi_3 + \int_{I_6} \varphi_3\varphi_3 & \int_{I_6} \varphi_4\varphi_3 \\
0 & 0 & \int_{I_6} \varphi_3\varphi_4 & \int_{I_6} \varphi_4\varphi_4 + \int_{I_7} \varphi_4\varphi_4
\end{pmatrix}
\end{aligned} \tag{2.18}
$$

Statt 112 Integrale müssen nur noch 16 berechnet werden. Wenn man diesen Schritt verallgemeinert, dann bedeutet das eine Verringerung der Zeitkomplexität für das Integrieren von $O\left(n^2\right)$ auf $O\left(n\right)$, wenn n die Anzahl an Shape-Funktionen ist.

Die Struktur der Matrix erlaubt zwei verschiedene Integrationsstrategien. Die erste Strategie ist, die Einträge der Matrix nacheinander zu berechnen. Diese Strategie hat den Vorteil, dass die Schreibzugriffe auf die Matrix und damit die Berechnung der Einträge ohne Probleme parallelisierbar ist. Der Nachteil dieser Integrationsstrategie liegt darin, dass bei naiver Implementierung die dünne Besetzungsstruktur der Matrix nicht ausgenutzt wird und dadurch die Berechnung wesentlich länger dauert. Prototyp 1 (Abschnitt 3.4) implementiert diese Integrationsstrategie und zeigt, wie die Nachteile verringert werden können.

Die zweite Integrationsstrategie iteriert über alle Integrationszellen und wertet direkt alle Integrale auf der Integrationszelle aus. Dadurch werden beim Integrieren mehrere Einträge der Matrix verändert, aber es werden garantiert nur Integrale berechnet, die einen Beitrag zu einem Matrixeintrag liefern. Diese Strategie wird in Prototyp 2 (Abschnitt 3.5) implementiert.

2.4 Nitsche-Methode für Randwerte

Nitsche schlägt in [Nit71] für die Einführung von Randwerten in Poisson-Probleme folgende Bilinearform und Linearform vor:

$$
\begin{aligned}
a\left(u,v\right) &= \int_\Omega \nabla u \nabla v - \int_{\partial\Omega} \left(\nabla u \cdot n\right) v - \int_{\partial\Omega} u \left(\nabla v \cdot n\right) + \beta \int_{\partial\Omega} uv \\
l\left(v\right) &= \int_\Omega fv - \int_{\partial\Omega} g \left(\nabla v \cdot n\right) + \beta \int_{\partial\Omega} gv
\end{aligned} \tag{2.19}
$$

Die Bilinearform und Linearform aus der schwachen Formulierung des Poisson-Problems für Finite Elemente werden durch partielle Integration gewonnen:

$$(2.20)\quad -\int_{\Omega} \Delta u v = \int_{\Omega} \nabla u \nabla v - \int_{\partial\Omega} (\nabla u \cdot n)\, v = \int_{\Omega} f v$$

Für klassische Finite Elemente wird in der Regel angenommen, dass der Raum so konstruiert ist, dass $\int_{\partial\Omega} \nabla u v = 0$ gilt. Für die in in den beiden Prototypen konstruierten Räume trifft das aber nicht zu. Das heißt die Bilinearform wird asymmetrisch, wodurch in die Voraussetzung einer symmetrisch, positiv definiten Bilinearform zur Minimierung des Funktionals $J(v)$ aus Abschnitt 2.1 verletzt ist. Die Symmetrie kann unter Zuhilfenahme der Randbedingungen wiederhergestellt werden:

$$(2.21)\quad \int_{\Omega} \nabla u \nabla v - \int_{\partial\Omega} (\nabla u \cdot n)\, v - \int_{\partial\Omega} u\, (\nabla v \cdot n) = \int_{\Omega} f v - \int_{\partial\Omega} g\, (\nabla v \cdot n)$$

Hierbei ist zu beachten, dass auf der rechten Seite wegen der Beziehung $u = g$ auf $\partial\Omega$ die Randwerte für $u|_{\partial\Omega}$ eingesetzt wurden.

Durch Regularisierung mit dem Term $\beta \int_{\partial\Omega} uv$ kann die positive Definitheit der Bilinearform wiederhergestellt werden. Dies ist ohne Probleme möglich, da $\int_{\partial\Omega} uv$ linear abhängig zu $\int_{\partial\Omega} (\nabla u \cdot n)\, v$ ist. Damit ergeben sich die von Nitsche vorgeschlagene Bilinearform und Linearform:

$$(2.22)\quad \int_{\Omega} \nabla u \nabla v - \int_{\partial\Omega} (\nabla u \cdot n)\, v - \int_{\partial\Omega} u\, (\nabla v \cdot n) + \beta \int_{\partial\Omega} uv = \int_{\Omega} f v - \int_{\partial\Omega} g\, (\nabla v \cdot n) + \beta \int_{\partial\Omega} gv$$

Obwohl der Regularisierungsparameter β nur ausreichend groß gewählt werden muss, kann die Wahl eines optimalen Regularisierungsparameters zur signifikanten Beschleunigung der Lösungsberechnung mit dem Konjugierte Gradienten Verfahren (CG-Verfahren) führen. Wie dieser Parameter ohne großen Mehraufwand berechnet werden kann, beschreibt [Scho5].

2.5 Sattelpunktprobleme

Sattelpunktprobleme entstehen, wenn an Variationsprobleme der Form

$$(2.23)\quad J(v) = \frac{1}{2} a(v, v) - f(v) \to \min v \in S_h$$

weitere Bedingungen gestellt werden. Dadurch ergibt sich eine Variationsaufgabe unter Nebenbedingungen (siehe [Bra07]):

$$\begin{aligned} J(v) &= \frac{1}{2} a(v,v) - f(v) \to \min v \in S_h \\ b(v,\mu) &= \langle g, \mu \rangle \text{ für } \mu \in M_h \end{aligned} \tag{2.24}$$

Die Lösung (u, λ) dieses Problems lässt sich durch

$$\begin{aligned} a(u,v) + b(v,\lambda) &= f(v) \\ b(u,\mu) &= \langle g, \mu \rangle \end{aligned} \tag{2.25}$$

bestimmen. λ wird dabei als Lagrangescher Parameter bezeichnet. Das Gleichungssystem ergibt sich wie in Abschnitt 2.1 dargestellt und sieht wie folgt aus:

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \cdot \begin{pmatrix} u \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{2.26}$$

mit

$$\begin{aligned} A_{i,j} &= a(\phi_j, \phi_i) \\ B_{i,j} &= b(\phi_j, \psi_i) \\ f_i &= f(\phi_i) \\ g_i &= g(\psi_i) \end{aligned} \tag{2.27}$$

Für die Prototypen waren zwei Sattelpunktprobleme von Bedeutung, das Poisson-Sattelpunktproblem und das Stokes-Sattelpunktproblem.

2.5.1 Poisson-Sattelpunktproblem

Aus dem Poisson-Problem

$$\begin{aligned} -\Delta u &= f \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{2.28}$$

wird durch Umformen in ein System aus Differentialgleichungen ein Sattelpunktproblem:

$$\begin{aligned} \sigma &= \nabla u \quad \text{auf } \Omega \\ \operatorname{div}\sigma &= -f \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{2.29}$$

mit $\Delta u = \operatorname{div}\sigma = \operatorname{div}\nabla u$ (siehe [Brao7]). Die dazugehörigen Bilinearformen und Linearformen sehen wie folgt aus:

$$\begin{aligned} a\left(\phi_j, \phi_i\right) &= \int_\Omega \phi_j \phi_i \\ b\left(\phi_j, \psi_i\right) &= \int_\Omega \operatorname{div}\phi_i \psi_i - \int_{\partial\Omega} \left(\phi_i \cdot n\right) \psi_i \\ f\left(\phi_i\right) &= 0 \\ g\left(\psi_i\right) &= -\int_\Omega f \psi_i \end{aligned} \tag{2.30}$$

2.5.2 Stokes-Sattelpunktproblem

Das Stokes-Problem beschreibt die stationäre Strömung in einer unendlich zähen, inkompressiblen Flüssigkeit.

$$\begin{aligned} -\Delta u + \nabla p &= f \quad \text{auf } \Omega \\ \operatorname{div} u &= 0 \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{2.31}$$

Die dazugehörigen Bilinearformen und Linearformen sehen wie folgt aus:

$$\begin{aligned} a\left(\phi_j, \phi_i\right) &= \int_\Omega \nabla\phi_j \nabla\phi_i \\ b\left(\phi_j, \psi_i\right) &= \int_\Omega \operatorname{div}\phi_i \psi_i - \int_{\partial\Omega} \left(\phi_i \cdot n\right) \psi_i \\ f\left(\phi_i\right) &= \int_\Omega f \phi_i \\ g\left(\psi_i\right) &= 0 \end{aligned} \tag{2.32}$$

Algorithmus 2.1 Gaußsches Eliminationsverfahren

```
procedure SolveGauss(A,b,x)
    n ← sizeof(A)
    for i = 1 → n do                                   // Vorwärtselimination
        for j = i + 1 → n do
            f ← A_{j,i} / A_{i,i}
            for m = i → n do
                A_{j,m} ← A_{j,m} − f · A_{i,m}
            end for
            b_j ← b_j − f · b_i
        end for
    end for
    for i = n → 1 do                                   // Rücksubstitution
        z ← b_i
        for j = i + 1 → n do
            z ← z − x_j · A_{i,j}
        end for
        x_i ← z / A_{i,i}
    end for
end procedure
```

2.6 Lösung

Die Lösung des aufgestellten Gleichungssystems ist der letzte Schritt bei der Lösung von Partition of Unity Problemen. Da es sich bei den aufgestellten Gleichungssystemen um lineare Gleichungssysteme handelt, kann hier prinzipiell jeder dafür bekannte Lösungsalgorithmus verwendet werden. In Prototyp 1 wird dafür das Gaußsche Eliminationsverfahren verwendet (siehe Algorithmus 2.1). Der Vorteil dieses Algorithmus liegt darin, dass nur die Invertierbarkeit der Matrix vorausgesetzt wird. Der Nachteil liegt in der hohen Zeitkomplexität der Berechnung. Die Lösung eines Gleichungssystems mit dem Gaußschen Eliminationsverfahren benötigt für voll besetzte Matrizen $O\left(n^3\right)$ Operationen.

Das Konjugierte Gradienten Verfahren (CG-Verfahren, siehe Algorithmus 2.2) nähert sich iterativ an die Lösung für das Gleichungssystem an. Dabei wird die quadratische Form

$$E\left(x\right) = \frac{1}{2}\left\langle Ax, x\right\rangle - \left\langle b, x\right\rangle \tag{2.33}$$

Algorithmus 2.2 Konjugierte Gradienten Verfahren (CG)

procedure SolveCG(A,b,x,ϵ,i_{max})
  $r \leftarrow f - Ax$ // x ist mit einer Anfangslösung belegt, in der Regel 0
  $d \leftarrow r$
  $i \leftarrow 0$
  while $i < i_{max} \wedge \|x\| > \epsilon$ **do**
    $\alpha \leftarrow \frac{d^T \cdot r}{d^T A d}$
    $x \leftarrow x - \alpha d$
    $r_{Next} \leftarrow r - \alpha A d$
    $\beta \leftarrow \frac{r_{Next}^T \cdot r_{Next}}{r^T \cdot r}$
    $d \leftarrow r_{Next} + \beta d$
    if $i \bmod 100 = 0$ **then** // alle 100 Iterationen das Residuum neu berechnen,
      $r \leftarrow f - Ax$ // um Rundungsfehler zu beseitigen
    else
      $r \leftarrow r_{Next}$
    end if
    $i \leftarrow i + 1$
  end while
end procedure

Algorithmus 2.3 Uzawa-Algorithmus

procedure Uzawa(A,B,f,u,p,ϵ,i_{max},ϵ_{CG},$i_{max,CG}$)
  $g \leftarrow f - B^T p$ // p ist mit einer Anfangslösung belegt, in der Regel 0
  SolveCG(A,g,u,ϵ_{CG},$i_{max,CG}$)
  $d \leftarrow Bu$
  $i \leftarrow 0$
  while $i < i_{max} \wedge \|d\| > \epsilon$ **do**
    $p \leftarrow p + d$
    $g \leftarrow f - B^T p$
    SolveCG(A,g,u,ϵ_{CG},$i_{max,CG}$)
    $d \leftarrow Bu$
    $i \leftarrow i + 1$
  end while
end procedure

minimiert, wobei $\langle a, b \rangle = \sum_i a_i b_i$ das Standardskalarprodukt ist. Die dafür verwendete Methode ist ähnlich dem Gradientenabstieg zur Minimumsuche für Funktionen. Der Gradient der quadratischen Form an einem Punkt x_k ist

$$\nabla E(x_k) = Ax_k - b = -r_k \tag{2.34}$$

Die initiale Abstiegsrichtung d_k wird für den CG-Algorithmus in Richtung des Residuums r_k gewählt, wobei dieses durch einen Faktor α_k skaliert wird. Damit das Minimum zu der quadratischen Form existiert muss die Matrix A symmetrisch und positiv definit sein. Dies kann für die hier vorgestellten Partition of Unity Probleme sichergestellt werden. Dadurch können Gleichungssysteme mit der Zeitkomplexität einer Matrixmultiplikation gelöst werden. Diese ist für vollbesetzte Matrizen auch $O(n^2)$, allerdings für dünn besetzte Matrizen, in der Form, wie sie bei den hier vorgestellten Partition of Unity Problemen auftreten, liegt die Zeitkomplexität nur noch bei $O(n)$. Voraussetzung dafür ist, dass die Besetzungsstärke bei $O(n)$ Nicht-Null-Einträgen liegt. Die Anzahl der benötigten Iterationen liegt bei $O(n)$. Dadurch hat der gesamte CG-Algorithmus für dünn besetzte Matrizen eine Zeitkomplexität von $O(n^2)$.

Der Uzawa-Algorithmus 2.3 ist ein Iterationsverfahren für die Lösung von Sattelpunktproblemen der Form 2.26. Dafür wird zunächst ein Wert für den Lagrangeschen Parameter berechnet und die Lösung u aktualisiert:

$$\begin{aligned} Au_k &= f - B^T \lambda_{k-1} \\ \lambda_k &= \lambda_{k-1} + \alpha (Bu_k - g) \end{aligned} \tag{2.35}$$

Für λ_0 wird ein Anfangswert vorgegeben. Für das Stokes-Problem wird das Uzawa-Verfahren auch als Druckiterationsverfahren bezeichnet, weil der Druck als Lagrangescher Parameter iterativ verfeinert wird und die Geschwindigkeit zu einem Druck berechnet wird.

3 Implementierung

Während der Diplomarbeit sind zwei Prototypen entstanden, die die Lösung von Differentialgleichungen mittels der Partition of Unity Methode umsetzen. Speziell geht es um die numerische Lösung der stationären, inkompressiblen Stokes-Gleichung. Die Prototypen sind so konzipiert worden, dass sie in der Studienarbeit [Lei12] verwendet werden konnten.

Als Vorgehensmodell wurde eine iterative Entwicklung mit sehr kurzen Iterationen, in der Regel in der Länge einer Woche, gewählt. Dadurch konnten Probleme bei der Implementierung der Patition of Unity Methode schnell entdeckt und behoben werden und die Priorisierung von Features innerhalb kurzer Zeit angepasst werden.

3.1 Teststrategie

Für die vollständige Abdeckung aller Testfälle mit Unittests wäre ein Aufwand nötig gewesen, der den Zeitrahmen der Diplomarbeit gesprengt hätte. Daher wurden vollständige Unittests zunächst nur für kleine Klassen und nichttriviale Methoden geschrieben. Alle weiteren Funktionen wurden mittels Systemtest durch vordefinierte Probleme, zu denen eine bekannte Lösung existiert, getestet. Nachdem im Systemtest ein Fehler gefunden wurde, wurde ein Testfall zur Reproduktion des Fehlers erstellt und dieser behoben. Ein Fehler im Systemtest ist dadurch definiert, dass die berechnete Lösung von der bekannten Lösung stärker als der zu erwartende numerische Fehler abweicht. Dafür wurde unter anderem das Visualisierungswerkzeug ParaView [par] benutzt.

3.2 Codequalität

Eine wichtige Anforderung an den Quellcode war, dass dieser zum Verständnis für andere Entwickler bei der Implementierung der Partition of Unity Methode dienen soll. Daher war es wichtig eine gute Lesbarkeit und Simplizität [LL07] zu erreichen. Dafür wurden lange Bezeichner und ein bewährter Styleguide gewählt und regelmäßig Refactorings durchgeführt.

Die Bezeichner für Klassen, Methoden und Parameter wurden so gewählt, dass sie die Beschreibung der Implementierung in [Sch05] wieder spiegeln, um die Verständlichkeit zu erhöhen.

Listing 3.1 Unterschiede zum Qt-Styleguide

```
// einzelne bedingte Anweisungen werden auch in geschweifte Klammern gesetzt
if (i % 2 == 0) {
    j++;
} else {
    k++;
}

// if direkt nach else auf einer Zeile
if (a == b) {
    return 0;
} else if (a < b) {
    return -1;
} else {
    return 1;
}
```

Als Styleguide wurde der im Qt-Projekt eingesetzte Styleguide [qts] mit einer kleinen Änderungen übernommen. Geschweifte Klammern werden auch bei einzeiligen bedingten Anweisungen gesetzt, außer bei einem `if`, das direkt auf ein `else` folgt, dann werden beide Anweisungen in die selbe Zeile geschrieben (siehe Listing 3.1).

Trotz regelmäßigem Refactoring konnten nicht alle Codeduplikate vermieden werden, weil die Verallgemeinerung der Algorithmen für alle auftretenden Fälle zu kompliziert gewesen wäre und die Verständlichkeit nicht wesentlich erhöht hätte.

3.3 Dokumentation

Als Dokumentation dienen diese Diplomarbeit und das Implementierungspaper von Schweitzer [Sch05].

3.4 Prototyp 1

Ziel bei der Entwicklung des ersten Prototyps war es, die Probleme zu erkennen, die bei der Implementierung der Partition of Unity Methode auftreten können, und Lösungen für diese zu finden. Anforderungen an den Prototyp waren die korrekte Lösung von Problemen basierend auf der stationären, inkompressiblen Stokes-Gleichung auf einem rechteckigen Gebiet mit Dirichlet-Randwerten und ohne Enrichments. Der lokale Ansatzraum und die Patches sollten austauschbar sein.

3.4.1 Architektur

Die Berechnung der Lösung lässt sich in drei übergeordnete Schritte unterteilen:

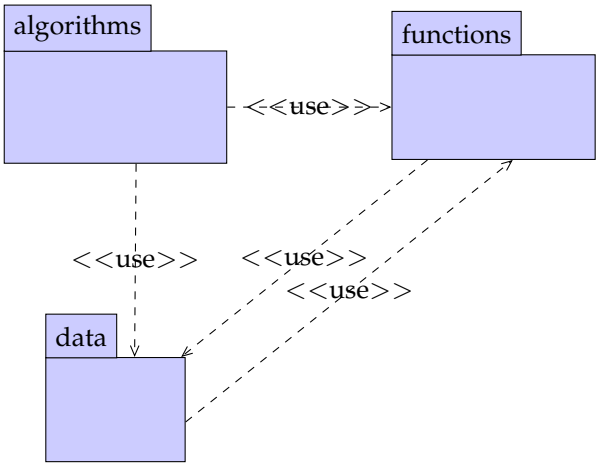

Abbildung 3.1: Organisation der Objekte in Algorithmen, Funktionen und Daten und deren Beziehung untereinander.

1. **Integrationszellen berechnen:** Zu einem bereitgestellten Cover werden die Gewichts- und PUM-Funktionen aufgestellt und die benötigten Integrationszellen berechnet.
2. **Matrix und Rechte Seite aufstellen:** Berechnung der Einträge der Matrix und des Rechte-Seite-Vektors durch Auswertung der Bilinear- und Linearform.
3. **Gleichungssystem lösen:** Lösung des aufgestellten Gleichungssystem.

Diese Schritte auszuführen wird im Prototyp 1 jedem Algorithmus selbst auferlegt. Diese Struktur ist aus dem iterativen Vorgehen so gewachsen und wurde so beibehalten, weil ein Refactoring mehr Aufwand, als eine erneute Implementierung in einem zweiten Prototypen, bedeutet hätte.

Abbildung 3.1 zeigt die allgemeine Organisation der Objekte in die drei Gruppen Algorithmen, Daten und Funktionen. Zu Algorithmen gehören alle Lösungsalgorithmen für Partition of Unity Probleme, Lösungsalgorithmen für Gleichungssysteme und Quadraturregeln. Zu Funktionen gehören alle mathematischen Funktionen und ähnliche Objekte, die an Koordinaten ausgewertet werden müssen. Zu Daten gehören alle Container, die Informationen für Algorithmen oder Funktionen bereitstellen, das sind beispielsweise Intervalle, Integrationszellen und Lösungen.

Die Entscheidung dafür Funktionen als Objekte, anstatt als Zeiger auf Funktionen, darzustellen hat den Vorteil, dass diese um weitere Informationen angereichert werden können und Funktionalität durch Vererbung weitergereicht werden kann.

Für die Partition of Unity Lösungsalgorithmen kommt das Entwurfsmuster Template Method [GHJV95] zur Anwendung, das die Reihenfolge der einzelnen Schritte eines Algorithmus vorgibt, aber nicht festlegt, wie diese Schritte ihr Ziel erreichen (siehe Abbildung 3.2).

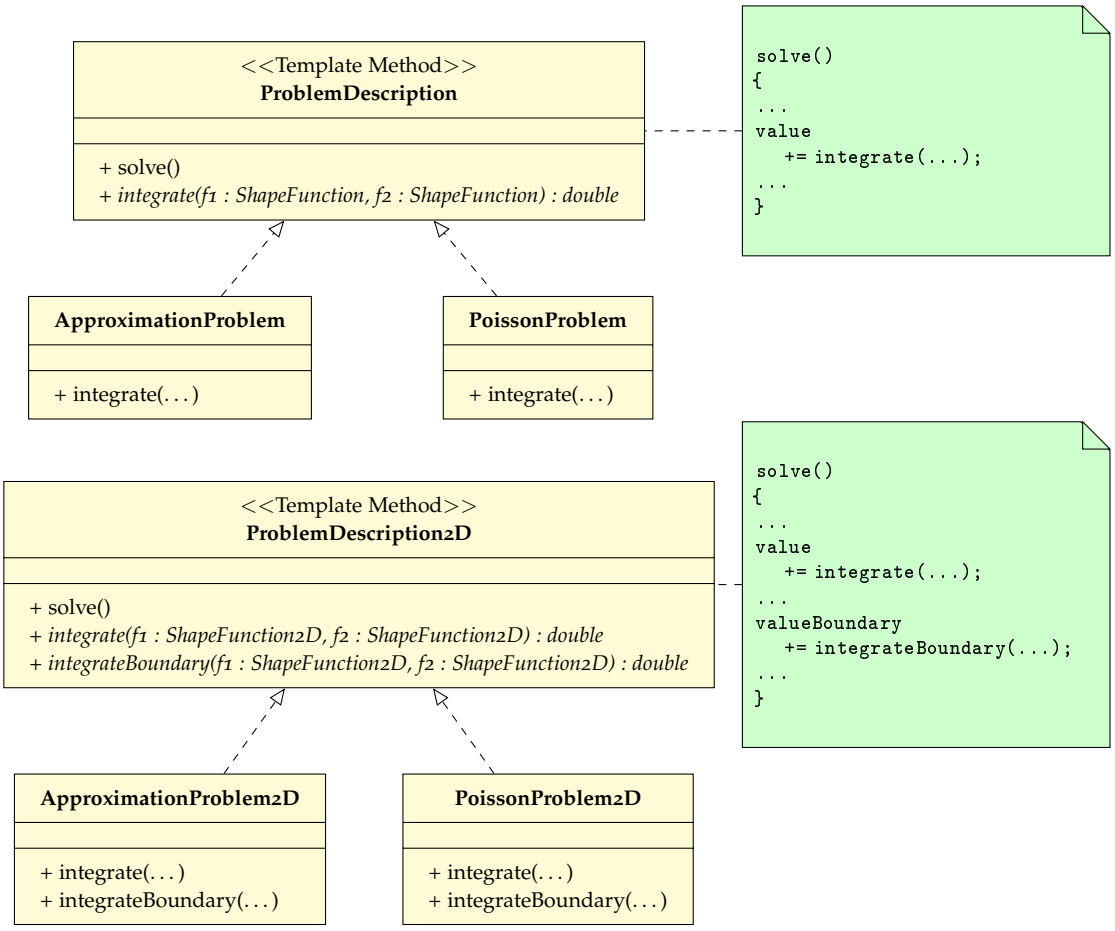

Abbildung 3.2: UML-Klassendiagramm für das verwendete Design-Pattern Template Method. Für Sattelpunktprobleme sehen die Klassen analog aus, mit dem einzigen Unterschied, dass es eigene Methoden für die Integration auf dem primären und sekundären Cover gibt, um die beiden Bilinearformen abbilden zu können.

Um sicherzustellen, dass Objekte eine bestimmt Funktionalität bereitstellen, werden an Stellen, wo der eigentliche Typ des Objektes egal ist, Interfaces definiert. Das betrifft Funktionen, Quadraturregeln und die Ausgabe der Lösung. Für Funktionen sind die Interfaces `Function` und `Funtion2D`, für Quadraturregeln die Interfaces `QuadratureRule` und `QuadratureRule2D` und für die Ausgabe der Lösung das Interface `SolutionPrinter` definiert (Abbildung 3.3).

3.4.2 Iteration 1: Grundgerüst und Approximationsproblem

In der ersten Iteration wurden alle grundlegenden Klassen zur Realisierung eines Partition of Unity Algorithmus implementiert, dazu gehören Patches, Cover, Integrationszellen, Gewichts-

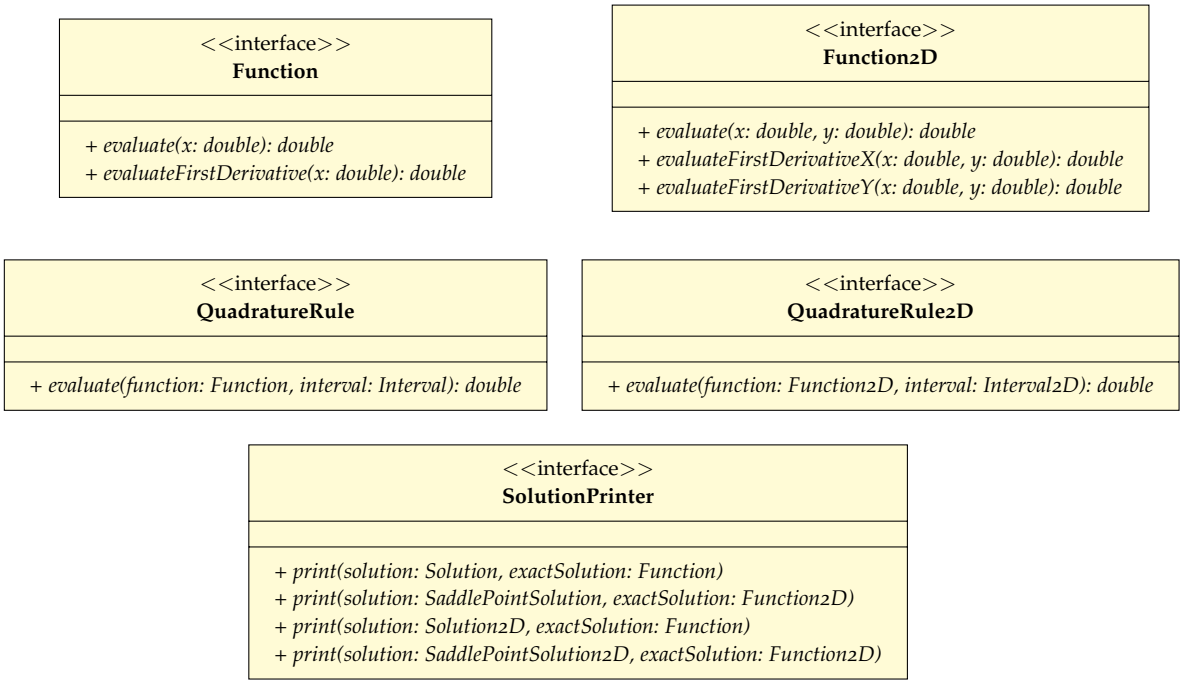

Abbildung 3.3: Vereinfachte Darstellung der Interfaces `Function`, `Function2D`, `QuadratureRule`, `QuadratureRule2D` und `SolutionPrinter` aus Prototyp 1.

, PUM- und Shape-Funktionen. Als Gewichtsfunktionen wurden lineare Splines gewählt (siehe Abschnitt 2.2.1). Als einfachen Test wurde die Lösung von Approximationsproblemen der Form

$$u = f \quad \text{auf } \Omega \tag{3.1}$$

implementiert. Die dafür benötigte Bilinearform und Linearform sind:

$$\begin{aligned} a(\phi_i, \phi_j) &= \int_\Omega \phi_i \phi_j \\ l(\phi_i) &= \int_\Omega f \phi_i \end{aligned} \tag{3.2}$$

Eine Lösung für ein Approximationsproblem zeigt Abbildung 3.4.

Approximationsprobleme zu lösen hat den Vorteil, dass diese zunächst ohne Randwerte auskommen, da weitere Bedingungen der Form $u = g$ auf $\partial\Omega$ redundant wären. Dies kann

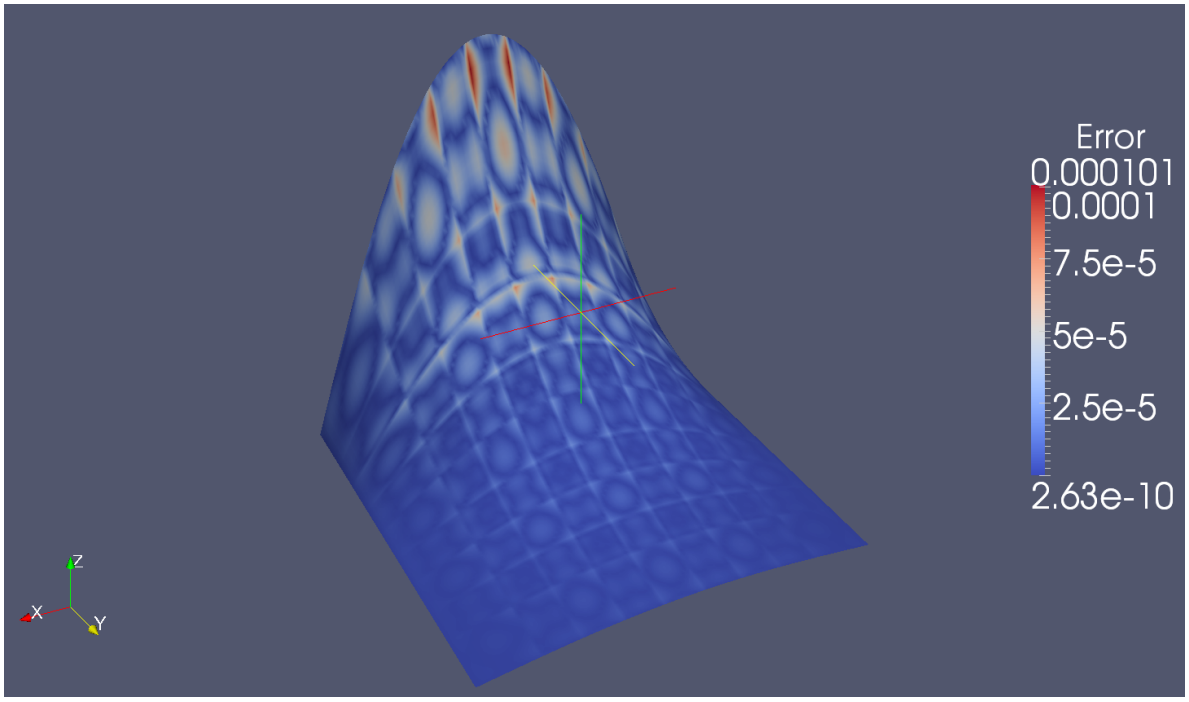

Abbildung 3.4: Approximation der Funktion $f(x,y) = \sin(\pi x)\, e^{-\pi y}$ mit einem kubischen Ansatzraum ($\{1, x, x^2, x^3\} \times \{1, y, y^2, y^3\}$) und einem regulären Cover aus 5×5 Patches auf dem Gebiet $\Omega = [0,1] \times [0,1]$.

aber dafür verwendet werden um die Funktion der Randwerte zu testen, indem das Problem zu

$$\begin{aligned} u &= f \quad \text{auf } \Omega \\ u &= f \quad \text{auf } \partial\Omega \end{aligned} \tag{3.3}$$

erweitert wird.

Cover

Prototyp 1 implementiert ein reguläres Cover. Neben der Eigenschaft, dass es das Gebiet überdeckt, bietet es noch weitere Vorteile. Die Anzahl der Integrationszellen wird durch die regelmäßige Anordnung der Patches möglichst gering gehalten und mit einem Stretch von 1.5 ergeben sich bestmögliche Patches, bei denen das Flat-Top-Intervall maximiert und gleichzeitig der Gradient am Überlapp minimiert wird. Die Algorithmen 3.1 und 3.2 zeigen, wie ein solches reguläres Cover erzeugt wird. Ein zweidimensionales reguläres Cover aus vier mal vier Patches, das das Gebiet $\Omega = [-1,1] \times [-1,1]$ überdeckt, ist in Abbildung 3.5 gezeigt.

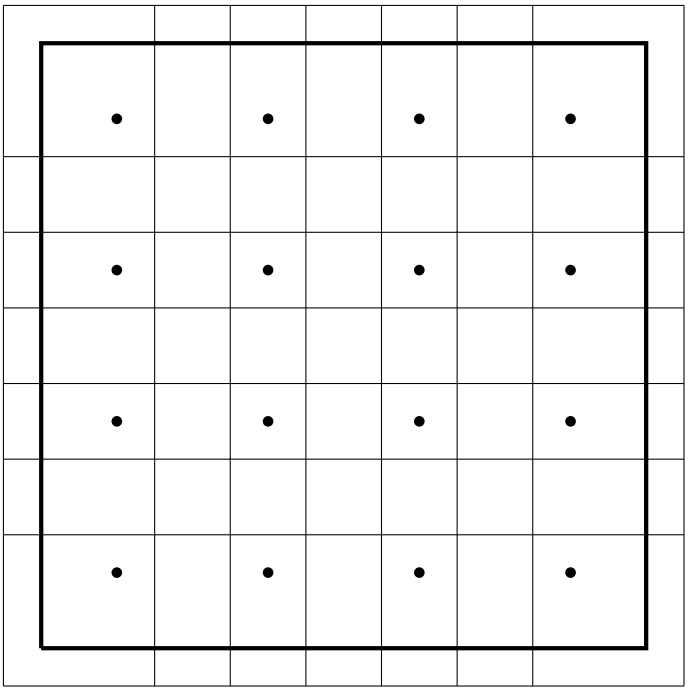

Abbildung 3.5: Zweidimensionales reguläres Cover mit 4×4 Patches über dem Gebiet $\Omega = [-1, 1] \times [-1, 1]$. Punkte kennzeichnen die Mittelpunkte der Patches.

Algorithmus 3.1 Erstellung eines eindimensionalen regulären Covers

function RegularCover($\Omega = [a, b]$,n,α)
  $Patches \leftarrow \{\}$
  $\Delta x \leftarrow \frac{b-a}{n}$ // Schrittweite
  $r \leftarrow \alpha \frac{\Delta x}{2}$ // Patchgröße (Radius)
  $x_M \leftarrow a + \frac{\Delta x}{2}$
  for $i = 1 \rightarrow n$ **do**
    $Patches \leftarrow Patches \cup \{Patch\,(x_M - r, x_M + r)\}$
    $x_M \leftarrow x_M + \Delta x$
  end for
  return *Patches*
end function

Algorithmus 3.2 Erstellung eines zweidimensionalen regulären Covers

function RegularCover2D($\Omega = [a,b] \times [c,d], n_x, n_y, \alpha_x, \alpha_y$)
  $Patches \leftarrow \{\}$
  $\Delta x \leftarrow \frac{b-a}{n_x}$ // Schrittweite
  $\Delta y \leftarrow \frac{d-c}{n_y}$
  $r_x \leftarrow \alpha_x \frac{\Delta x}{2}$ // Patchgröße (Radius)
  $r_y \leftarrow \alpha_y \frac{\Delta y}{2}$
  $y_M \leftarrow c + \frac{\Delta y}{2}$
  for $i = 1 \rightarrow n_x$ **do**
    $x_M \leftarrow a + \frac{\Delta x}{2}$
    for $j = 1 \rightarrow n_y$ **do**
      $Patches \leftarrow Patches \cup \left\{ Patch2D\left(x_M - r_x, x_M + r_x, y_M - r_y, y_M + r_y\right)\right\}$
      $x_M \leftarrow x_M + \Delta x$
    end for
    $y_M \leftarrow y_M + \Delta y$
  end for
  return *Patches*
end function

Algorithmus 3.3 Berechnung der Integrationszellen zu einem eindimensionalen Cover

function IntegrationCells($P = \{\omega_1, \ldots, \omega_n\}$)
  $I \leftarrow \{\}$
  for $\omega_i \in P$ **do**
    $r \leftarrow \omega_i$ // Rest zur Berechnung des Flat-Top-Intervalls
    for $\omega_j \in P$ **do**
      if $i \neq j \wedge r \cap \omega_j \neq \emptyset$ **then**
        $I \leftarrow I \cup \left\{r \cap \omega_j\right\}$
        $r \leftarrow r \setminus \omega_j$
      end if
    end for
    $I \leftarrow I \cup \{r\}$
  end for
  return I
end function

Integrationszellen

Algorithmus 3.3 zeigt, wie Integrationszellen im Eindimensionalen aus einer Menge Patches berechnet werden. Das grundlegende Vorgehen dabei ist, dass von einem Patch alle Überlappe mit anderen Patches weggeschnitten werden und der verbleibende Rest das Flat-Top-Intervall des Patches wird. Hierbei wird vorausgesetzt, dass die Gewichtsfunktionen

Algorithmus 3.4 Berechnung der Integrationszellen zu einem zweidimensionalen Cover

```
function IntegrationCells2D(P = {ω_1, ..., ω_n})
    P_x = {}
    P_y = {}
    for ω_i ∈ P do                          // Trennung der einzelnen Dimensionen
        P_x ← P_x ∪ {ω_{i,x}}
        P_y ← P_y ∪ {ω_{i,y}}
    end for
    I_x ← IntegrationCells(P_x)
    I_y ← IntegrationCells(P_y)
    I ← {}
    for J ∈ I_x do                          // zweidimensionale Integrationszellen sind
        for K ∈ I_y do                      // das Tensorprodukt der eindimensionalen
            I ← I ∪ {J × K}                 // Integrationszellen
        end for
    end for
    return I
end function
```

keine weitere Unterteilung der Patches implizieren. Dies ist bei den verwendeten linearen Splines der Fall, wenn deren Spitze innerhalb des Flat-Top-Intervalls liegt.

Algorithmus 3.4 zeigt, wie aus den in Algorithmus 3.3 konstruierten eindimensionalen Integrationszellen zweidimensionale Integrationszellen erzeugt werden können. Es werden zunächst Integrationszellen für jede Dimension berechnet und dann die zweidimensionalen Integrationszellen als Tensorprodukt beider Mengen erstellt. Damit das funktioniert, müssen die Patches ebenfalls aus einem Tensorprodukt zweier Mengen eindimensionaler Intervalle konstruiert worden sein. Dies wird durch ein reguläres Cover sichergestellt.

Randwerte

Dirichlet-Randwerte werden im Prototyp 1 durch die Lösung eines Approximationsproblems auf dem Rand in die Rechnung mit einbezogen ($u = g$ auf $\partial\Omega$). Dazu werden im eindimensionalen die konstante Funktion $\psi_0 = x^0 = 1$ und im zweidimensionalen alle lokalen Approximationsfunktionen bei denen einer der beiden Faktoren die konstante Funktion ist auf allen Randpatches benutzt. Alle anderen lokalen Ansatzfunktionen werden so transformiert, dass diese auf dem Rand null sind (siehe Abbildung 3.6). Dadurch werden alle Randintegrale für diese Funktionen null und sind nicht mehr für die Lösung relevant.

Aufstellen der Matrix und der rechten Seite

Die Matrix wird in Prototyp 1 Eintrag für Eintrag erzeugt. Das heißt zu jeder Kombination zweier Shape-Funktionen wird die Bilinearform ausgewertet und in den jeweiligen Matrixein-

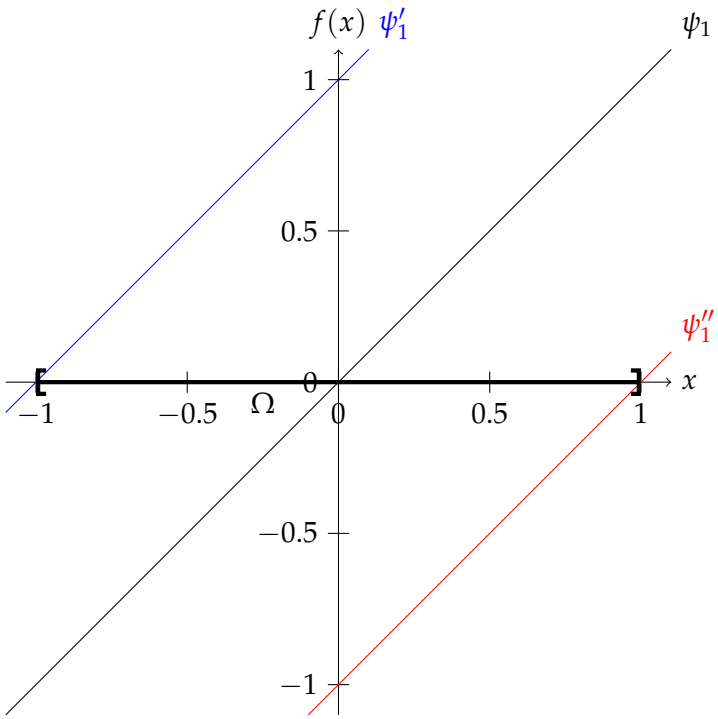

Abbildung 3.6: Transformation der lokalen Ansatzfunktion $\psi_1 = x$ auf den linken Rand $x = -1.0$ von Ω (ψ_1') und auf den rechten Rand $x = 1.0$ von Ω (ψ_1'').

trag gespeichert. Diese Methode lässt sich einfach parallelisieren, weil für jede Auswertung genau ein Matrixeintrag geschrieben wird und Lesezugriffe in der Regel beliebig parallel durchgeführt werden können. Der Nachteil ist, dass ohne weitere Maßnahmen sehr viele Nulleinträge unnötig berechnet werden, obwohl diese schon aus der Nachbarschaft der Patches bestimmt werden könnten. Mehr dazu in Abschnitt 3.4.6.

3.4.3 Iteration 2: Poisson-Problem

In der zweiten Iteration wurde die Lösung von Poisson-Problemen der Art

$$\begin{aligned} -\Delta u &= f \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{3.4}$$

Algorithmus 3.5 Aufstellen der Matrix und der rechten Seite für ein PUM-Problem

procedure AssembleMatrixAndRightHandSide(Ω, $S = \{\phi_{1,0}, \ldots, \phi_{n,m}\}$, $P = \{\omega_1, \ldots, \omega_n\}$, $I = \{I_1, \ldots, I_l\}$, f, g, F)
  for $\phi_{i,p} \in S$ **do**
    if $\omega_i \cap \partial\Omega \neq \emptyset \wedge p = 0$ **then** // Randpatch und konstante Ansatzfunktion
      for $I_k \in I$ **do** // bestimmen den Randwert
        $A_{i,i} \leftarrow \int_{\partial\Omega \cap I_k} \phi_{i,p}|_{\partial\Omega} \phi_{i,p}|_{\partial\Omega}$
        $f_i \leftarrow \int_{\partial\Omega \cap I_k} g\phi_{i,p}|_{\partial\Omega}$
      end for
    else // Freiheitsgrad
      for $\phi_{j,q} \in S$ **do**
        for $I_k \in I$ **do**
          $A_{i,j} \leftarrow \int_{I_k} F\left(\phi_{i,p}, \phi_{j,q}\right)$
        end for
      end for
      for $I_k \in I$ **do**
        $f_i \leftarrow \int_{I_k} f\phi_{i,p}$
      end for
    end if
  end for
end procedure

implementiert. Für Poisson-Probleme müssen Randwerte zur eindeutigen Bestimmung der Lösung vorgegeben werden. Die dafür benötigten Bilinearformen und Linearformen sehen wie folgt aus:

$$\begin{aligned} a\left(\phi_i, \phi_j\right) &= \int_\Omega \nabla\phi_i \nabla\phi_j \\ l\left(\phi_i\right) &= \int_\Omega f\phi_i \end{aligned} \tag{3.5}$$

für Freiheitsgrade und

$$\begin{aligned} a\left(\phi_i, \phi_j\right) &= \int_{\partial\Omega} \phi_i\phi_j \\ l\left(\phi_i\right) &= \int_{\partial\Omega} g\phi_i \end{aligned} \tag{3.6}$$

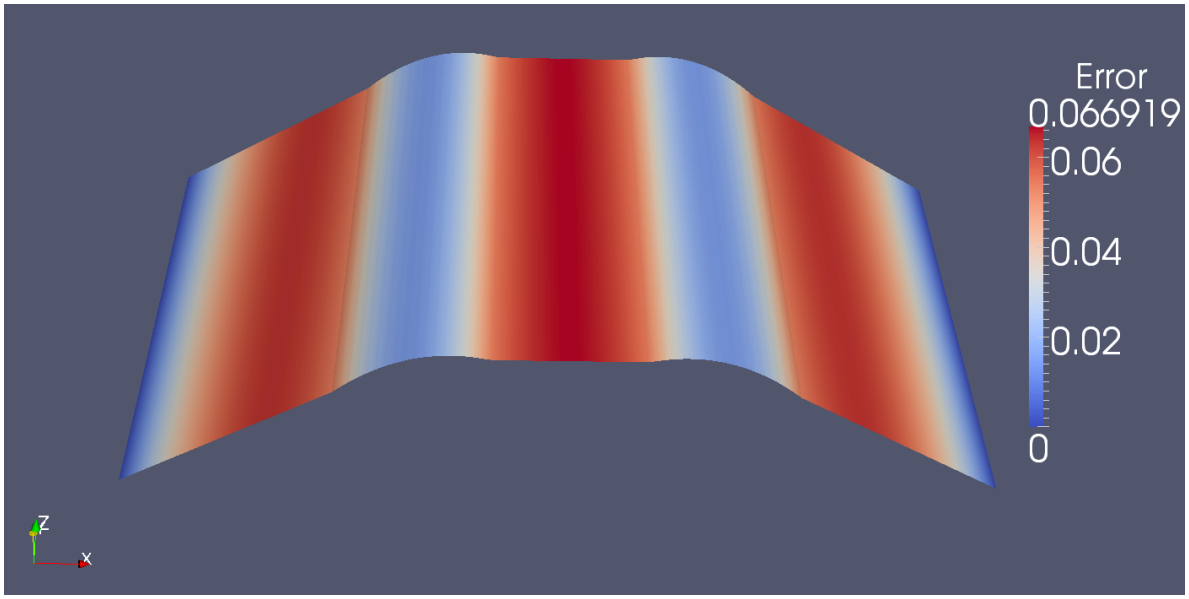

Abbildung 3.7: Lösung des eindimensionalen Poisson-Problems mit linearem Ansatzraum und einem regulären Cover aus 3 Patches. Die Lösung ist nach Größe des Fehlers zur exakten Lösung eingefärbt: Blau zeigt an, dass die Lösung dort gut bis sehr gut approximiert wird, und Rot, dass die Lösung dort stark abweicht.

für Randfunktionen. Die Bilinearform und Linearform für Randwerte definiert ein Approximationsproblem auf dem Rand für alle Funktionen, die dort nicht null sind (siehe Abschnitt 3.4.2).

Der Einfluss der Sonderbehandlung der Randwerte auf die Gestalt der Matrix ist in folgendem Beispiel zu sehen:

$$\begin{aligned} -\Delta u &= 1 \quad \text{auf } [-1,1] \\ u &= 1 \quad \text{auf } \partial[-1,1] \end{aligned} \tag{3.7}$$

mit dem lokalen Ansatzraum $\{1, x\}$ und einem regulären Cover mit 3 Patches und Stretch $\alpha = 1.5$. Daraus ergibt sich folgende Matrix:

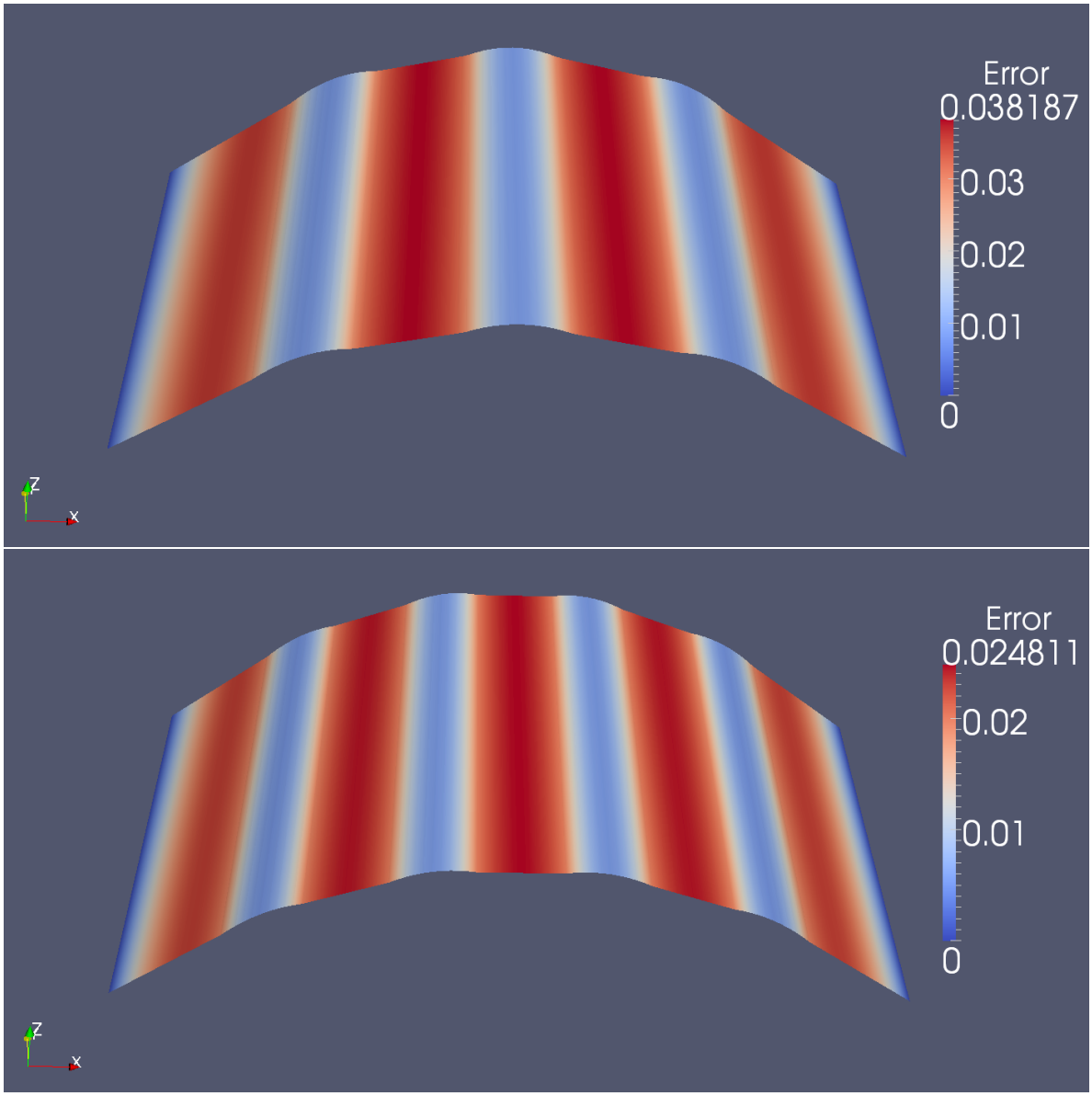

Abbildung 3.8: Vergrößerung des Cover auf 4 (oben) bzw. 5 (unten) Patches zur Verbesserung der Lösung.

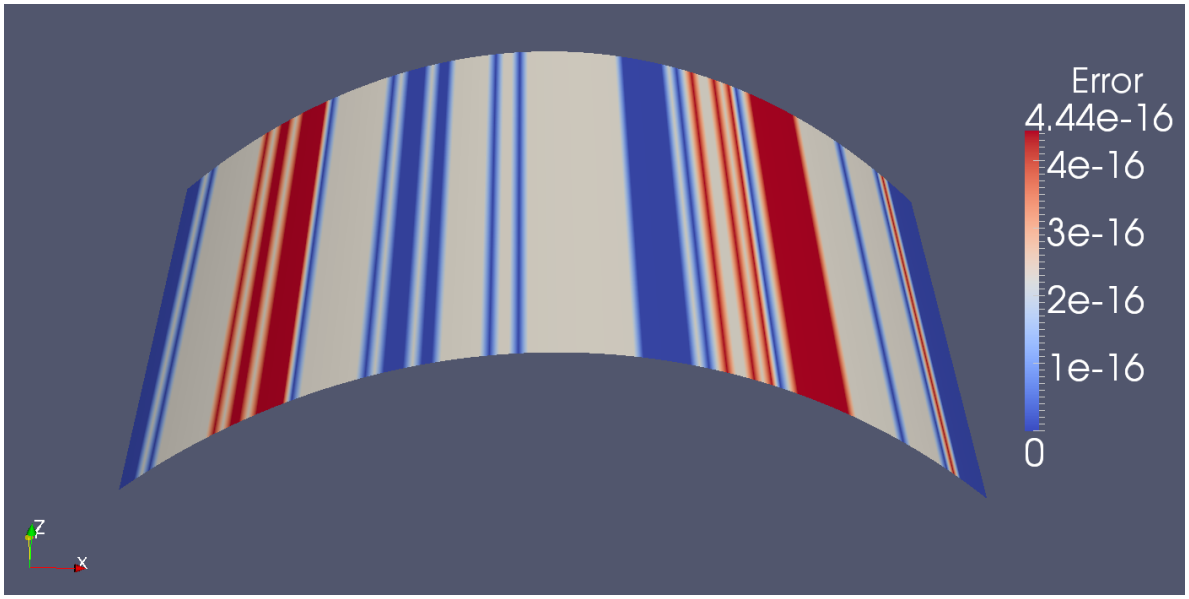

Abbildung 3.9: Vergrößerung des lokalen Ansatzraums auf $\{1, x, x^2\}$ zur Verbesserung der Lösung.

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
1.5 & 1.36111 & -1.5 & 0.138889 & 0 & 0 \\
-3 & -1.5 & 6 & -5.55112 \cdot 10^{-16} & -3 & 1.5 \\
0.5 & 0.138889 & -5.55112 \cdot 10^{-16} & 0.722222 & -0.5 & 0.138889 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1.5 & 0.138889 & -1.5 & 1.36111
\end{pmatrix}
\cdot
\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{pmatrix}
= \tag{3.8}
$$

$$
\begin{pmatrix}
1 \\
0.226852 \\
0.666667 \\
-7.28584 \cdot 10^{-17} \\
1 \\
-0.226852
\end{pmatrix}
$$

Die Symmetrie der Poisson-Matrix wird durch das Approximationsproblem auf dem Rand durchbrochen. Dieser Effekt ist deutlich in Zeile 1 und 5 der Matrix zu sehen. Im Beispiel ist das Approximationsproblem auf dem Rand durch $u_0\phi_{1,0}(-1) = 1$ und $u_4\phi_{3,0}(1) = 1$ gegeben, wie an den Einträgen der Matrix zu erkennen ist. Die Lösung dieses Gleichungssystems ist in Abbildung 3.7 dargestellt. Die Abbildung 3.8 zeigt, dass sich der numerische Fehler mir zunehmender Anzahl Patches verringert. In Abbildung 3.9 ist zu sehen, dass sich der Fehler durch einen verbesserten lokalen Approximationsraum signifikant verringern kann.

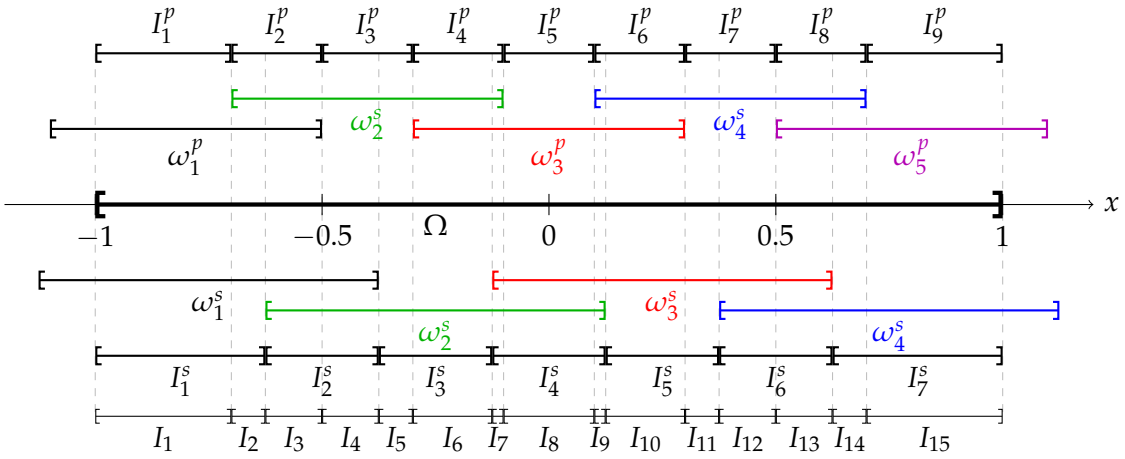

Abbildung 3.10: Integrationszellen für ein Sattelpunktproblem über $\Omega = [-1, 1]$ mit einem primären Cover aus fünf Patches und einem sekundären Cover aus vier Patches.

3.4.4 Iteration 3: Sattelpunktprobleme

In der dritten Iteration wurde die Lösung von Sattelpunktproblemen im Prototyp 1 implementiert. Zunächst nur die Lösung des Poisson-Sattelpunktproblems, weil hier gegen die Lösung aus dem normalen Poisson-Problem verglichen werden konnte. Um aus dem normalen Poisson-Problem ein Sattelpunktproblem zu erzeugen, wird der Laplace-Operator wie folgt umgeformt:

$$\Delta u = \operatorname{div} \nabla u = -f \tag{3.9}$$

Jetzt setzen wir $\nabla u = \sigma$ und formen die Gleichung 3.9 in ein System um:

$$\begin{aligned} \nabla u &= \sigma \\ \operatorname{div} \sigma &= -f \end{aligned} \tag{3.10}$$

Daraus ergibt sich folgendes Sattelpunktproblem:

$$\begin{aligned} (\sigma, \tau) + (\operatorname{div} \tau, u) &= 0 && \text{auf } \Omega \\ (\operatorname{div} \sigma, v) &= -(f, v) && \text{auf } \Omega \\ u &= g && \text{auf } \partial\Omega \end{aligned} \tag{3.11}$$

Die unterschiedlichen Räume für u und σ werden durch zwei Cover erreicht. Das Cover für σ wird als primäres Cover bezeichnet und das Cover für u als sekundäres Cover. Im Prototyp 1 wird aus beiden Bilinearformen eine Matrix erzeugt. Für den eindimensionalen Fall konnte einfach der Algorithmus für normale Partition of Unity Probleme erweitert werden. Für den zweidimensionalen Fall mussten vektorwertige Funktionen für das primäre Cover eingeführt werden, damit die Divergenz korrekt berechnet werden kann. Die neuen vektorwertigen Shape-Funktionen sehen wie folgt aus:

$$S^p = \left\{ \begin{pmatrix} \phi^p_{1,0} \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} \phi^p_{n_p,m_p} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \phi^p_{1,0} \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ \phi^p_{n_p,m_p} \end{pmatrix} \right\} \tag{3.12}$$

Da in der Bilinearform $b(\cdot,\cdot)$ Integrale über Produkte aus Shape-Funktionen aus beiden Covern ausgewertet werden müssen, müssen auch die Integrationszellen angepasst werden. Hierzu werden einfach die Integrationszellen für das primäre Cover an den Grenzen der Integrationszellen für das sekundäre Cover zerschnitten (siehe Abbildung 3.10).

Die für die Lösung des Poisson-Sattelpunktproblems in der Implementierung benötigten Bilinearformen und Linearformen sehen wie folgt aus:

$$\begin{aligned} a(\phi_i,\phi_j) &= \int_\Omega \phi_i\phi_j \\ b(\phi_i,\phi_j) &= \int_\Omega \operatorname{div}\phi_i\phi_j + \int_{\partial\Omega} (\phi_i \cdot n)\,\phi_j \\ l(\phi_i) &= \int_\Omega f\phi_i \end{aligned} \tag{3.13}$$

für Freiheitsgrade und

$$\begin{aligned} a(\phi_i,\phi_j) &= \int_{\partial\Omega} \phi_i\phi_j \\ b(\phi_i,\phi_j) &= \int_{\partial\Omega} \phi_i\phi_j \\ l(\phi_i) &= \int_{\partial\Omega} g\phi_i \end{aligned} \tag{3.14}$$

für Randfunktionen. Der Randterm in der Bilinearform $b(\cdot,\cdot)$ für Freiheitsgrade entsteht durch die partielle Integration und fällt nicht weg, weil die Konstruktion der verwendeten PUM-Funktionen so ist, dass diese auf dem Rand nicht null sind, wie beispielsweise in [Bra07] angenommen.

3.4.5 Iteration 4: Stokes-Sattelpunktproblem

Für die Implementierung von Stokes-Sattelpunktproblem der Form

$$\begin{aligned} -\Delta u + \nabla p &= f \quad \text{auf } \Omega \\ \text{div} u &= 0 \quad \text{auf } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned} \tag{3.15}$$

wurde eine komplette Iteration durchgeführt. Die dafür benötigte Bilinearform und Linearform sind:

$$\begin{aligned} a\left(\phi_i, \phi_j\right) &= \int_\Omega \nabla\phi_i \nabla\phi_j \\ b\left(\phi_i, \phi_j\right) &= \int_\Omega \text{div}\, \phi_i \phi_j + \int_{\partial\Omega} \left(\phi_i \cdot n\right) \phi_j \\ l\left(\phi_i\right) &= \int_\Omega f\phi_i \end{aligned} \tag{3.16}$$

für Freiheitsgrade und

$$\begin{aligned} a\left(\phi_i, \phi_j\right) &= \int_{\partial\Omega} \phi_i \phi_j \\ b\left(\phi_i, \phi_j\right) &= \int_{\partial\Omega} \phi_i \phi_j \\ l\left(\phi_i\right) &= \int_{\partial\Omega} g\phi_i \end{aligned} \tag{3.17}$$

für Randfunktionen.

3.4.6 Iteration 5: Performance Optimierungen

Die Lösung von Sattelpunktproblemen erwies sich mit den bis hier hin implementierten Algorithmen als zu langsam. Die folgenden Optimierungen dienen dazu, die benötigte Rechenzeit zu verringern.

Algorithmus 3.6 Optimiertes Aufstellen der Matrix und der rechten Seite für ein PUM-Problem

procedure AssembleMatrixAndRightHandSide(Ω, $S = \{\phi_{1,0}, \ldots, \phi_{n,m}\}$, $P = \{\omega_1, \ldots, \omega_n\}$, $I = \{I_1, \ldots, I_l\}$, f, g, F)
  for $\phi_{i,p} \in S$ **do**
    if $\omega_i \cap \partial\Omega \neq \emptyset \wedge p = 0$ **then** // Randpatch und konstante Ansatzfunktion
      for $I_k \in \{I_k \in I : I_k \cap \omega_i \neq \emptyset\}$ **do** // bestimmen den Randwert
        $A_{i,i} \leftarrow \int_{\partial\Omega \cap I_k} \phi_{i,p}|_{\partial\Omega} \phi_{i,p}|_{\partial\Omega}$
        $f_i \leftarrow \int_{\partial\Omega \cap I_k} g\phi_{i,p}|_{\partial\Omega}$
      end for
    else // Freiheitsgrad
      for $\phi_{j,q} \in S$ **do**
        for $I_k \in \{I_k \in I : I_k \cap \omega_i \neq \emptyset\} \cap \{I_k \in I : I_k \cap \omega_j \neq \emptyset\}$ **do**
          $A_{i,j} \leftarrow \int_{I_k} F\left(\phi_{i,p}, \phi_{j,q}\right)$
        end for
      end for
      for $I_k \in \{I_k \in I : I_k \cap \omega_i \neq \emptyset\}$ **do**
        $f_i \leftarrow \int_{I_k} f\phi_{i,p}$
      end for
    end if
  end for
end procedure

Funktionen kennen Integrationszellen

Bei der Erstellung der Integrationszellen wird die Information $\{I_k \in I : I_k \cap \omega_i \neq \emptyset\}$ in den Funktionsobjekten gespeichert und bei der Integration wieder abgerufen. Die Berechnung der Information beinhaltet kein Mehraufwand, da diese für die Erstellung der Integrationszellen schon benötigt wird. Dadurch verringert sich der Aufwand bei der Integration bei Überprüfung, auf welchen Integrationszellen ein Integral ausgewertet werden muss, auf drei für ein eindimensionales reguläres Cover und auf neun für ein zweidimensionales reguläres Cover, von ursprünglich n nötigen Überprüfungen. Bei Sattelpunktproblemen ist die Zahl der Überprüfungen etwas höher, weil sich durch die Verwendung von zwei Covern die Zahl der Integrationszellen, auf denen eine Funktion lebt, erhöht. Der Aufwand für die Überprüfung bleibt aber trotzdem bei $O(1)$ statt bei $O(n)$. Algorithmus 3.6 zeigt die optimierte Erzeugung der Matrix und der rechten Seite.

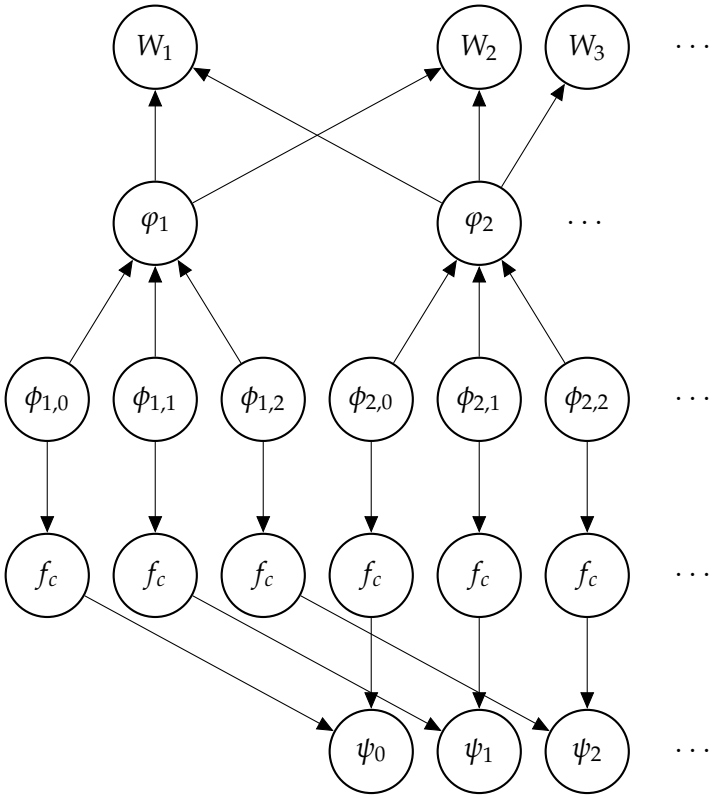

Abbildung 3.11: Verlinkung zwischen den einzelnen Funktionen für ein eindimensionales Problem. Alle PUM-Funktionen greifen auf eine gemeinsame Menge Gewichtsfunktionen zurück. Alle Shape-Funktionen greifen auf eine gemeinsame Menge PUM-Funktionen zurück. Lokale Ansatzfunktionen werden aus einer gemeinsamen Menge von den Shape-Funktionen nur transformiert referenziert.

Vorauswertung

Einen wesentlichen Geschwindigkeitsvorteil bringt die Vorberechnung der Funktionswerte. Abbildung 3.11 zeigt die Abhängigkeiten der Funktionen untereinander und ist wie folgt zu lesen: zur Auswertung der Funktion $\phi_{0,1}$ muss die Funktion φ_1 und dafür die Funktionen W_1 und W_2 ausgewertet werden. Die Matrix besteht aus Integralen der Form $\int \phi_i \phi_j$, das heißt jede Shape-Funktion taucht in n Integralen auf und davon in einem sogar zweimal. Davon sind viele der Produkte durch den lokalen Träger der Gewichtsfunktionen null. Übrig bleiben also im eindimensionalen Fall bis zu drei Integrale pro Shape-Funktion, in denen dieses ausgewertet werden muss (siehe Abschnitt 2.2). Wenn wir jetzt annehmen, dass der lokale Approximationsraum aus m Ansatzfunktionen besteht (in Abbildung 3.11 ist m drei), dann wird eine Gewichtsfunktion auf der Integrationszelle, auf der sich zwei Patches überlappen, $4m^2$ mal ausgewertet. Dieses lässt sich durch Vorberechnung des Werts auf eine Auswertung reduzieren (siehe 2.18).

Die Vorauswertung ist ohne Probleme möglich, da die Stellen, an denen eine Funktion ausgewertet werden muss, alleine durch das Integrationsintervall, also die Integrationszelle, und die verwendete Quadraturregel bestimmt sind. Diese Information ist vor der Integration vollständig vorhanden. Der Nachteil der Vorauswertung ist ein wesentlich erhöhter Speicherverbrauch. Es wird also Ersparnis an Rechenzeit mit einem erhöhten Verbrauch an Hauptspeicher bezahlt. Um den zusätzlichen Speicherverbrauch in Grenzen zu halten gibt es einige Maßnahmen, die aber aus Zeitgründen nicht in den Prototyp Einzug gehalten haben. Eine Maßnahme zeigt Prototyp 2, in dem die Integrationsstrategie geändert wurde.

Durch Einführung der Vorauswertung ergibt sich nun folgende Abfolge der Berechnungsschritte:

1. **Integrationszellen berechnen:** Zu einem bereitgestellten Cover werden die Gewichts- und PUM-Funktionen aufgestellt und die benötigten Integrationszellen berechnet.
2. **Vorauswertung der Shape-Funktionen:** Die Shape-Funktionen auf allen Integrationszellen an den Integrationspunkten auswerten und diese Werte zwischenspeichern.
3. **Matrix und Rechte Seite aufstellen:** Berechnung der Einträge der Matrix und des Rechte-Seite-Vektors durch Auswertung der Bilinear- und Linearform.
4. **Gleichungssystem lösen:** Lösung des aufgestellten Gleichungssystem.

Ausnutzung der dünnen Besetzungsstruktur der Matrizen

In Abschnitt 3.4.6 wurde gezeigt, wie sich die Berechnung mit der Information, auf welchen Integrationszellen eine PUM-Funktion lebt, beschleunigen lässt. Zur Auswertung der Integrale muss aber für zwei gegebene Funktionen immer noch überprüft werden, auf welchen Integrationszellen beide Funktionen leben:

$$\{I_k \in I : I_k \cap \omega_i \neq \emptyset\} \cap \{I_k \in I : I_k \cap \omega_j \neq \emptyset\} \tag{3.18}$$

Bei großen Problemen ist der Schnitt der beiden Mengen für fast alle Kombinationen von Patches leer. Das heißt, dass in den meisten Fällen der Aufwand für die Berechnung des Schnitts der beiden Mengen umsonst ist. Dieser Aufwand lässt sich durch eine einfache Überprüfung auf $\omega_i \cap \omega_j \neq \emptyset$ vermeiden, weil gilt:

$$\omega_i \cap \omega_j \neq \emptyset \Leftrightarrow \{I_k \in I : I_k \cap \omega_i \neq \emptyset\} \cap \{I_k \in I : I_k \cap \omega_j \neq \emptyset\} \neq \emptyset \tag{3.19}$$

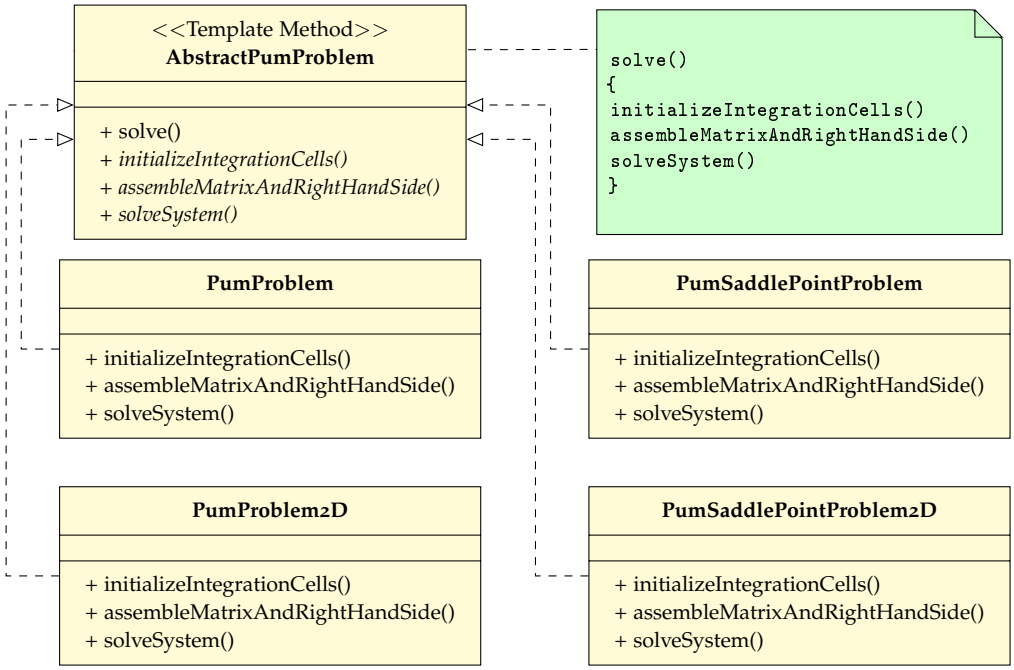

Abbildung 3.12: UML-Klassendiagramm für das verwendete Design-Pattern Template Method im Prototyp 2.

3.5 Prototyp 2

Ursprünglich sollten die Erkenntnisse aus dem Prototyp 1 zu dem Zeitpunkt, wo der Prototyp 2 begonnen wurde, in das Studienprojekt CraSS, das am Institut für Parallele und Verteilte Systeme in der Abteilung Simulation großer Systeme lief, einfließen. Allerdings war das Studienprojekt zu diesem Zeitpunkt noch nicht in einem solchen Zustand, dass dies möglich gewesen wäre. Ein weiteres Refactoring des Prototyp 1 war mit der Erkenntnis, dass eine andere Integrationsstrategie effizienter wäre, auch nicht mehr sinnvoll.

Ziel für den Prototyp 2 war es, Stokes-Sattelpunktprobleme effizienter als im Prototyp 1 lösen zu können, aber dabei trotzdem eine einfach verständliche Codebasis zu behalten. Außerdem sollten Erweiterungen um weitere Features möglich sein, auch wenn diese nicht im Rahmen dieser Diplomarbeit implementiert werden würden.

Zunächst musste die Funktionalität aus Prototyp 1 in den Prototyp 2 übernommen werden, dies geschah wie bei der Entwicklung von Prototyp 1 schrittweise, allerdings in nur einer Iteration. Nachdem die Funktionalität wiederhergestellt war wurde in kurzen Iterationen weiterentwickelt.

3.5.1 Architektur

In Prototyp 2 wurde die Verwendung des Entwurfsmusters weiter vertieft. Im Gegensatz zu Prototyp 1, wo nur die Integration als Template Method implementiert wurde, wird in Prototyp 2 der komplette Partition of Unity Algorithmus in einer Template Method abgebildet (siehe Abbildung 3.12). Als Alternative hätte auch ein Composite dienen können, dass die einzelnen Schritte des Algorithmus als Objekte hält, was aber wegen der starken Abhängigkeit der Schritte untereinander verworfen wurde.

Weiterhin wurde die Verständlichkeit durch stärkere Verwendung von Vererbung verbessert. So erbt beispielsweise `Domain` von `Interval`, statt eine Eigenschaft für das Intervall zu halten, wodurch das Domain-Objekt auch direkt als Intervall verwendet werden kann (`domain` gegenüber `domain.m_interval`). Gleiches gilt für Gewichtsfunktionen, Patches und Integrationszellen.

3.5.2 Dünn besetzte Matrizen

Im Prototyp 2 werden für die Speicherung der Matrizen dünn besetzte Matrizen verwendet. Dünn besetzte Matrizen speichern im Gegensatz zu vollbesetzten Matrizen nur die Einträge einer Matrix, die verschieden von null sind. Dadurch verringert sich der Speicherbedarf für die hier vorgestellten Partition of Unity Probleme von $O\left(n^2\right)$ auf $O\left(n\right)$, wenn n die Anzahl der Shape-Funktionen ist. Diese Speicherersparnis wird durch eine höhere Zugriffszeit auf einzelne Einträge der Matrix und einen höheren Aufwand bei der Erzeugung der Matrix erkauft, ermöglicht aber die effiziente Nutzung von iterativen Verfahren, wie dem CG-Verfahren zum Lösen von Gleichungssystemen und dem Uzawa-Verfahren zum Lösen von Sattelpunktproblemen.

Für dünn besetzten Matrizen wurde keine Bibliothek verwendet, weil der Aufwand für eine eigene Implementierung als so gering eingeschätzt wurde, dass die Einbindung einer Bibliothek mit hoher Wahrscheinlichkeit länger gedauert hätte. Dadurch konnte auch das Interface der Operationen für die dünn besetzten Matrizen genau an die im Programm vorhandenen Anforderungen angepasst werden. Diese Entscheidung hat im Verlauf der Entwicklung noch einen weiteren Vorteil gehabt, da die Matrix dadurch um weitere Funktionen angereichert werden konnte, die Zugriff auf die internen Datenstrukturen benötigen. Die wichtigste dieser Funktionen ist die Ausgabe einer Matrix im MATLAB-Format für dünn besetzte Matrizen.

Als Datenstruktur kommt eine etwas abgeänderte Implementierung der Block Compressed Row Storage (BCRS) Matrizen zum Einsatz. Der Unterschied zu klassischen BCRS Implementierungen ([Fag10]) besteht darin, dass die einzelnen Blöcke nicht über mehrere Zeilen der Matrix gehen, sondern immer genau eine Zeile groß sind. Außerdem werden die Informationen nicht in ein großes Array zusammen gepackt, sondern auf mehrere C++-Vektoren verteilt, um die Verständlichkeit zu erhöhen.

Die Matrix wird im Speicher zeilenweise abgelegt. Jede Zeile enthält Blöcke von aufeinanderfolgenden Nicht-Null-Einträgen. Zu jedem Block wird der Spaltenindex des ersten Eintrags gespeichert. Dadurch lässt sich der Spaltenindex jedes Elements in diesem Block durch

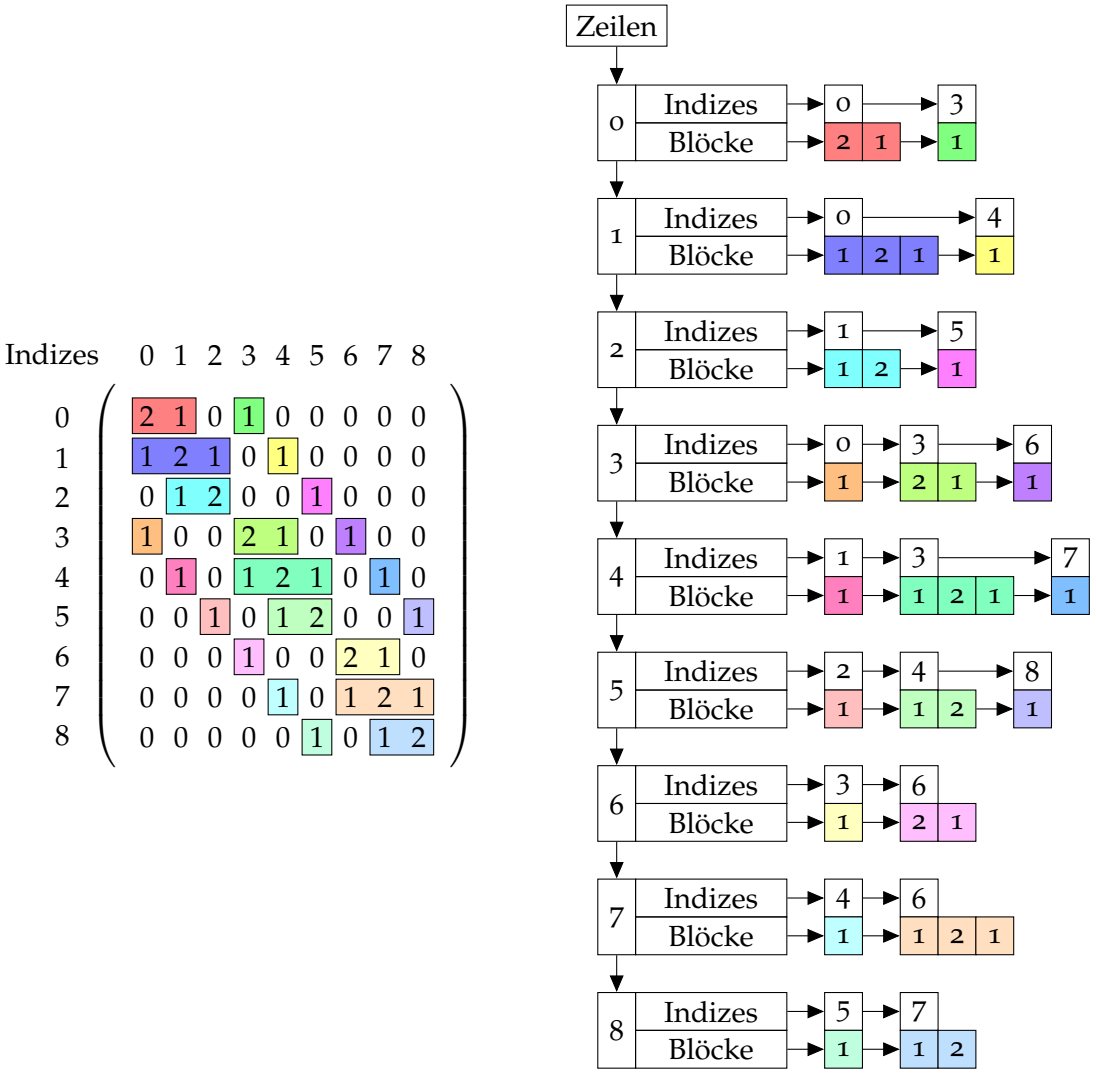

Abbildung 3.13: Matrix mit PUM-typischer Besetzungsstruktur und die Repräsentation in der Datenstruktur für dünne Matrizen.

Addition des Index des ersten Elements und des Index des Elements im Block bestimmen. Abbildung 3.13 zeigt eine solche Datenstruktur für eine repräsentative Beispielmatrix.

Die dünn besetzten Matrizen können ihre Größe zur Laufzeit dynamisch anpassen, das bedeutet, dass in einer Matrix neue Nicht-Null-Einträge alloziert werden können, nachdem auf den bestehenden Einträgen schon Rechnungen durchgeführt wurden. Accidential Zeros, also Nullen, die auf allozierte Einträge der Matrix geschrieben werden, bleiben erhalten, weil hier der Aufwand den Nutzen weit überwiegt.

Algorithmus 3.7 Berechnung der Integrationszellen in Prototyp 2

function IntegrationCells2($P = \{\omega_1, \ldots, \omega_n\}$)
  $I \leftarrow \{\}$
  for $\omega_i \in P$ **do**
    $I_{Temp} \leftarrow \omega_i$ // Zerschneide ω_i and den Rändern aller überlappenden Patches
    for $\omega_j \in P$ **do**
      if $i \neq j \wedge \omega_i \cap \omega_j \neq \emptyset$ **then**
        for $I_k \in I_{Temp}$ **do**
          $I_{Temp} \leftarrow I_{Temp} \setminus \{I_k\} \cup \{I_k \setminus \omega_j\} \cup \{I_k \cap \omega_j\}$
        end for
      end if
    end for
    for $I_k \in I_{Temp}$ **do**
      AddAdditionalInformation(I_k)
      $I \leftarrow I \cup I_k$
    end for
  end for
  return I
end function

Algorithmus 3.8 Aufstellen der Matrix und der rechten Seite für ein PUM-Problem in Prototyp 2

procedure AssembleMatrixAndRightHandSide2(Ω, $S = \{\phi_{1,0}, \ldots, \phi_{n,m}\}$, $P = \{\omega_1, \ldots, \omega_n\}$, $I = \{I_1, \ldots, I_r\}$, $a(\cdot,\cdot)$, $l(\cdot)$)
  for $I_k \in IntegrationCells$ **do**
    for $\phi_{i,p} \in S : I_k \cap \omega_i \neq \emptyset$ **do** // Diese Information wird beim Berechnen
      for $\phi_{j,q} \in S : I_k \cap \omega_j \neq \emptyset$ **do** // der Integrationszellen vorberechnet
        $A_{i,j} \leftarrow \int_{I_k} a_\Omega(\phi_{i,p}, \phi_{j,q}) + \int_{\partial\Omega \cap I_k} a_{\partial\Omega}(\phi_{i,p}, \phi_{j,q})$
      end for
      $f_i \leftarrow \int_{I_k} l_\Omega(\phi_{i,p}) + \int_{\partial\Omega \cap I_k} l_{\partial\Omega}(\phi_{i,p})$
    end for
  end for
end procedure

3.5.3 Integrationszellen

Die veränderte Integrationsstrategie setzt voraus, dass die Integrationszellen mehr Wissen über das Problem, insbesondere die Funktionen, die auf ihnen leben, haben. Außerdem wurde der Erzeugungsalgorithmus für die Integrationszellen angepasst, sodass dieser für beliebige Cover funktioniert. Ein weiterer Nebeneffekt ist, dass der Algorithmus auch Gebiete erlaubt, die aus mehr als einem Tensorproduktintervall bestehen.

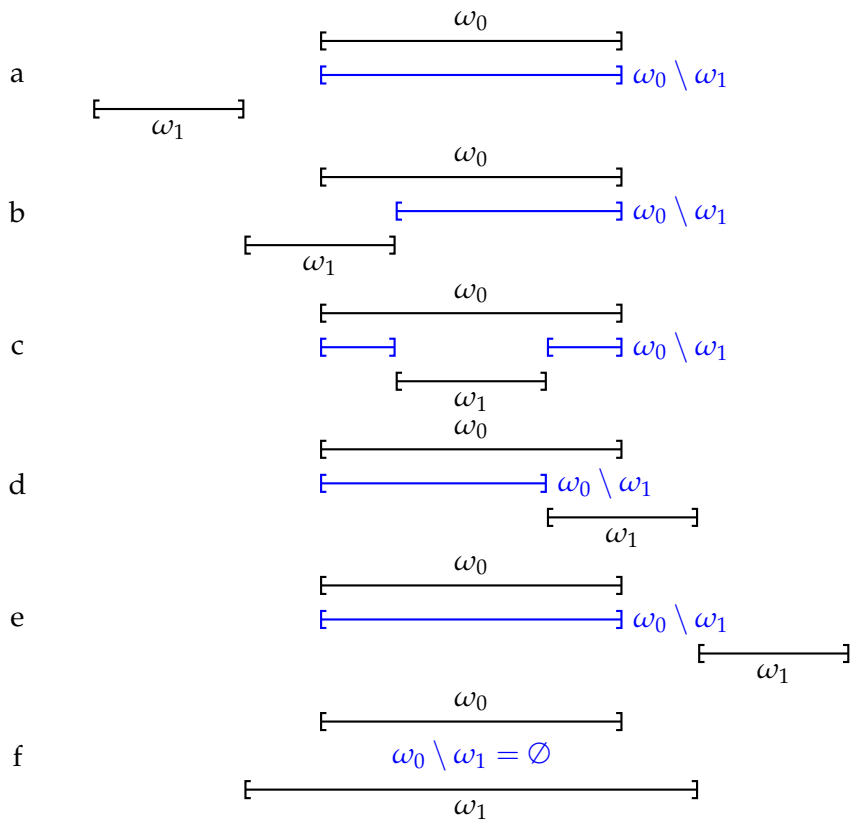

Abbildung 3.14: Fälle bei der Differenz zweier eindimensionaler Intervalle $\omega_0 \setminus \omega_1$.

Algorithmus 3.7 zeigt, wie unter diesen Voraussetzungen die Integrationszellen erstellt werden können. Die grundlegende Herangehensweise ist, dass die Patches an den Rändern ihrer Nachbarn zerschnitten werden. Dieser Algorithmus kann sowohl für eindimensionale als auch für zweidimensionale Partition of Unity Probleme verwendet werden. Dafür müssen für eindimensionale Intervalle und zweidimensionale Tensorproduktintervalle die Operationen Differenz ($I_k \setminus \omega_j$) und Schnitt ($I_k \cap \omega_j$) implementiert werden. Der Schnitt ist für beide Fälle trivial zu implementieren. Eine größere Herausforderung ist die Differenz. Die zu implementierenden Fälle sind in Abbildung 3.14 für den eindimensionalen Fall und Abbildung 3.15 für den zweidimensionalen Fall zu sehen. Zu Beachten ist, dass das Ergebnis der Differenz wieder eine Menge von Intervallen bzw. Tensorproduktintervallen sein muss, damit der Algorithmus wieder auf den Differenzrest angewendet werden kann.

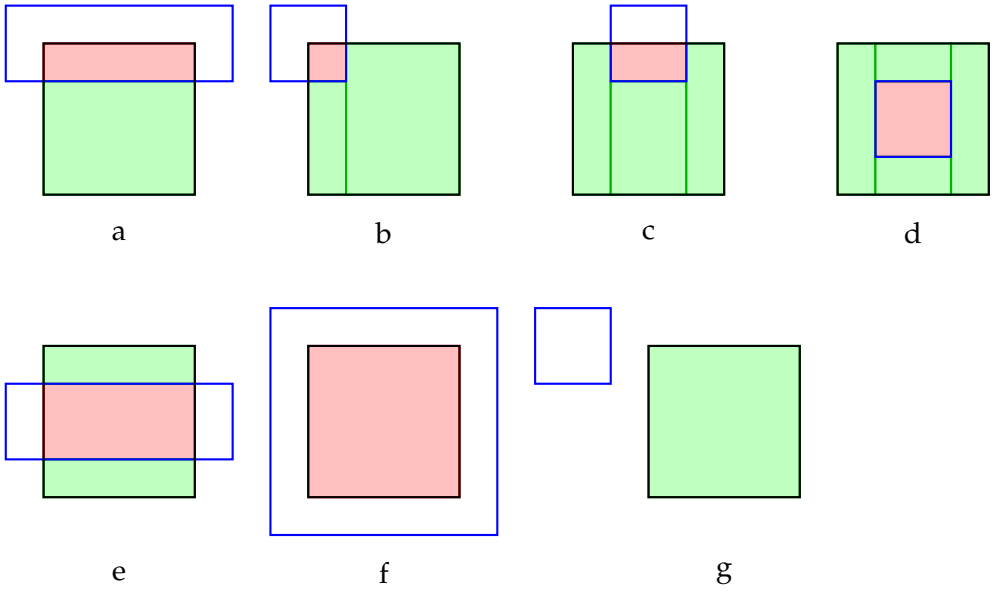

Abbildung 3.15: Referenzfälle beim Schnitt zweier zweidimensionaler Tensorproduktintervalle, alle weiteren Fälle lassen sich durch Spiegelung und Drehung aus diesen konstruieren. Die grün gekennzeichneten Flächen zeigen, wie der Differenzrest aus Tensorproduktintervallen erzeugt werden kann. Dies ist nötig, um den Differenzalgorithmus wieder auf den Differenzrest anwenden zu können.

Der Funktionsaufruf AddAdditionalInformation(I_k) reichert die Integrationszelle mit der Information an, welche Shape-Funktionen auf ihr leben. Diese Information würde zu dem Beispiel aus 2.2 wie folgt aussehen:

$$
\begin{aligned}
&I_1 : \{\phi_1\} \\
&I_2 : \{\phi_1, \phi_2\} \\
&I_3 : \{\phi_2\} \\
&I_4 : \{\phi_2, \phi_3\} \\
&I_5 : \{\phi_3\} \\
&I_6 : \{\phi_3, \phi_4\} \\
&I_7 : \{\phi_4\}
\end{aligned}
\tag{3.20}
$$

Dadurch lässt sich Algorithmus 3.8 zur effizienten Aufstellung der Matrix und rechten Seite benutzen.

3.5.4 Randintegrale

Prototyp 2 verwendet eine verbesserte Methode für die Berechnung von Randintegralen. Diese wurde nötig, weil der Prototyp komplexere Gebiete und damit komplexere Ränder unterstützen soll. Ein weiterer positiver Effekt ist, dass nicht mehr vier Randstücke eines Rechtecks einzeln unterschieden werden müssen, sondern alle Randstücke mit den gleichen Algorithmen behandelt werden können.

Integrale über ein Randstück lassen sich auf Integrale über das Intervall $[-1, 1]$ abbilden. Dies bringt Vorteile bei der Verwendung von Quadraturregeln, wie der Gauß-Quadratur, weil diese auch auf dem Intervall $[-1, 1]$ operieren. Für die Integration von zweidimensionalen Funktionen auf eindimensionalen Intervallen lassen sich Transformationen $x(t)$ und $y(t)$ definieren, die einen Punkt auf dem Intervall in den höherdimensionalen Raum abbilden. Dadurch lassen sich Randintegrale auf Shape-Funktionen durch

$$\int_{\partial\Omega} f(x,y) = \int_{t=-1}^{1} f(x(t), y(t)) \tag{3.21}$$

berechnen. Gleichzeitig können mit diesem Parameter komplexere Randbedingungen in der Form $g(t)$ vorgegeben werden.

3.5.5 Lösung

Durch die Verwendung von Datenstrukturen für dünn besetzte Matrizen, wird eine effiziente Lösung mit iterativen Verfahren ermöglicht. Daher wird für die Lösung der Gleichungssysteme in Prototyp 2 das CG-Verfahren implementiert. Für die Lösung von Sattelpunktproblemen wurde ein einfacher Uzawa-Algorithmus implementiert. Abbildung 3.16 zeigt, wie sich die Lösung während des Uzawa-Algorithmus iterativ verbessert.

3.5.6 Konfiguration

Zur Verbesserung der Bedienung des Programms, wurde für den Prototyp 2 die Möglichkeit implementiert, einige Parameter mit einer Konfigurationsdatei festzulegen. Dadurch muss das Programm bei Änderungen der Problemkonfiguration nicht erneut kompiliert werden. Zur besseren Erweiterbarkeit bestehen die Einträge in der Konfigurationsdatei aus Schlüssel-Wert-Paaren. Unbekannte Schlüssel werden ignoriert und alle Einstellungen werden mit Default-Werten vorbelegt. Dadurch wird vollständige Vorwärts- und Abwärtskompatibilität der Konfigurationsdateien gewährleistet.

Folgende Einstellungen lassen sich vornehmen:

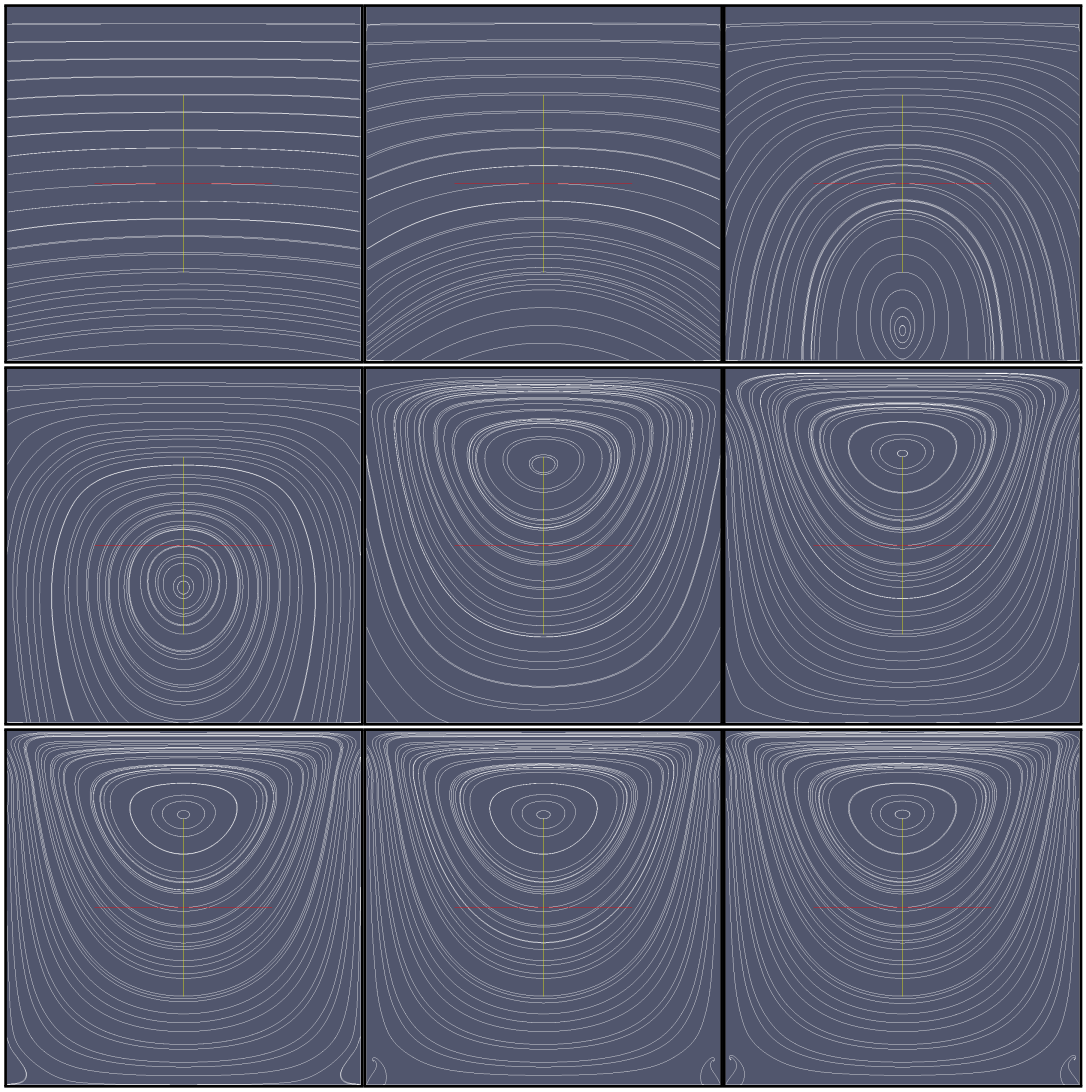

Abbildung 3.16: Iterative Annäherung an die Lösung mit dem Uzawa-Verfahren. Zu sehen ist die Lösung nach 1, 3, 7, 10, 50, 500, 15000, 50000, 100000 Iterationen (von links oben nach rechts unten).

- `primary_cover_number_patches_first_dimension` : Anzahl der Patches des primären Covers in x-Richtung
- `primary_cover_number_patches_second_dimension` : Anzahl der Patches des primären Covers in y-Richtung
- `primary_cover_stretch_first_dimension` : Stretch des primären Covers in x-Richtung
- `primary_cover_stretch_second_dimension` : Stretch des primären Covers in y-Richtung
- `secondary_cover_number_patches_first_dimension` : Anzahl der Patches des sekundären Covers in x-Richtung
- `secondary_cover_number_patches_second_dimension` : Anzahl der Patches des sekundären Covers in y-Richtung
- `secondary_cover_stretch_first_dimension` : Stretch des sekundären Covers in x-Richtung
- `secondary_cover_stretch_second_dimension` : Stretch des sekundären Covers in y-Richtung
- `primary_cover_local_approximation_space_degree` : Grad des lokalen Ansatzraums des primären Covers
- `secondary_cover_local_approximation_space_degree` : Grad des lokalen Ansatzraums des sekundären Covers
- `cg_tolerance` : Abbruch-Toleranz des CG-Algorithmus
- `cg_maximum_iterations` : Maximale Anzahl an Iterationen des CG-Algorithmus
- `uzawa_tolerance` : Abbruch-Toleranz des Uzawa-Algorithmus
- `uzawa_maximum_iterations` : Maximale Anzahl an Iterationen des Uzawa-Algorithmus
- `vtk_output_file` : Datei für die Ausgabe der VTK-Visualisierung der Lösung
- `matrix_output_folder` : Ausgabeverzeichnis für die Matrizen
- `solve` : Lösung des aufgestellten Gleichungssystems an-/abstellen
- `include` : Einbinden einer anderen Konfigurationsdatei
- `regularization_parameter` : Regularisierungsparameter
- `regularization_parameter_file` : Datei mit Regularisierungsparameter
- `number_of_vtk_samples` : Anzahl der Gitterpunkte für die VTK-Visualisierung

Das Einbinden anderer Konfigurationsdateien ermöglicht die Konfiguration in einer hierarchischen Struktur zu organisieren. Dadurch können beispielsweise größere Serien von Berechnungsläufen durchgeführt werden, ohne jedes Mal eine komplette Konfigurationsdatei erstellen zu müssen. Schlüssel können mehrfach definiert werden, dabei wird der zuletzt gelesene Wert gesetzt. Ist in der Konfigurationsdatei eine Datei für den Regularisierungsparameter angegeben und kann diese vom Programm gelesen werden, so erhält der

Regularisierungsparameter aus der Datei den Vorrang vor einem in der Konfigurationsdatei definierten.

Einige Einstellungen lassen sich nicht durch Einträge in der Konfigurationsdatei vornehmen, weil dies zu Aufwändig gewesen wäre. Dazu gehören:

- **Randwerte:** Die Definition von Randwerten ist sehr eng an die internen Datenstrukturen gekoppelt und würde die textuelle Definition von Funktionen in der Konfigurationsdatei voraussetzen.
- **Bilinearform, Linearform und Rechte Seite:** Die Definition der Bilinearform, Linearform und Rechte Seite würde die textuelle Definition von Funktionen in der Konfigurationsdatei voraussetzen.
- **Domain:** Die Definition der Randwerte ist eng mit der verwendeten Domain verbunden, sodass zwar die Domain in der Konfigurationsdatei hätte konfiguriert werden können, aber gleichzeitig die Definition der Randwerte hätte angepasst werden müssen, also kein Komfortgewinn vorhanden gewesen wäre.
- **Ausgabe Matrizen:** Die Ausgabe der Matrizen kann nicht konfiguriert werden, da externe Programme zur Berechnung des Regularisierungsparameters darauf angewiesen sind, dass die Matrizen vorhanden sind und diese in einem bestimmten Format vorliegen.

3.6 Vergleich der Prototypen

In diesem Abschnitt werden die beiden Prototypen hinsichtlich Codequalität, Geschwindigkeit, Speicherverbrauch und Erweiterbarkeit verglichen. Zum Vergleich der Codequalität soll die beispielhafte Definition eines Driven-Cavitiy-Stokes-Sattelpunktproblems in beiden Prototypen dienen.

3.6.1 Verständlichkeit

Die Listings 3.2 und 3.3 zeigen die Definition der lokalen Approximationsräume für das primäre und sekundäre Cover. In 3.2 wird der lokale Approximationsraum aus allgemeinen Polynomen, die zu Monomen definiert werden, zusammengesetzt, was etwas umständlicher ist, als die in 3.3 verwendeten Monome. Im zweiten Prototyp ist von vornherein vorgesehen die lokalen Approximationsräume für die beiden Cover unabhängig von einander zu behandeln.

Die Listings 3.4 und 3.5 zeigen die Definition der Domain und der Cover. Während in Prototyp 1 die Domain noch aus zwei eindimensionalen Intervallen zusammengesetzt wird, so wird in Prototyp 2 die Domain direkt durch die Ausdehnungen definiert. In Prototyp 1 müssen nach dem Erstellen der Cover noch die lokalen Approximationsräume den Patches zugewiesen werden, in Prototyp 2 passiert dies automatisch.

Listing 3.2 Beispielproblemdefinition im Prototyp 1 für das Driven-Cavity-Problem: Definition der lokalen Approximationsräume

```
std::vector<Function*> localApproximationSpace;

std::vector<double> constantCoefficients;
constantCoefficients.push_back(1);
localApproximationSpace.push_back(new PolynomialFunction(constantCoefficients, true));

std::vector<double> linearCoefficients;
linearCoefficients.push_back(1);
linearCoefficients.push_back(0);
localApproximationSpace.push_back(new PolynomialFunction(linearCoefficients, false));

std::vector<double> quadricCoefficients;
quadricCoefficients.push_back(1);
quadricCoefficients.push_back(0);
quadricCoefficients.push_back(0);
localApproximationSpace.push_back(new PolynomialFunction(quadricCoefficients, false));

std::vector<double> qubicCoefficients;
qubicCoefficients.push_back(1);
qubicCoefficients.push_back(0);
qubicCoefficients.push_back(0);
qubicCoefficients.push_back(0);
localApproximationSpace.push_back(new PolynomialFunction(qubicCoefficients, false));
```

Listing 3.3 Beispielproblemdefinition im Prototyp 2 für das Driven-Cavity-Problem: Definition der lokalen Approximationsräume

```
std::vector<const OnceDerivableFunction2D*> localApproximationSpace2d;
for (size_t i = 0; i <=
    Settings::currentSettings()->primaryCoverLocalApproximationSpaceDegree(); i++) {
   for (size_t j = 0; j <=
       Settings::currentSettings()->primaryCoverLocalApproximationSpaceDegree(); j++) {
      localApproximationSpace2d.push_back(
                new OnceDerivableTensorProductFunction2D<const OnceDerivableFunction>(
                   new MonomialFunction(i),
                   new MonomialFunction(j)));
   }
}

std::vector<const OnceDerivableFunction2D*> secondaryLocalApproximationSpace2d;
for (size_t i = 0; i <=
    Settings::currentSettings()->secondaryCoverLocalApproximationSpaceDegree(); i++) {
   for (size_t j = 0; j <=
       Settings::currentSettings()->secondaryCoverLocalApproximationSpaceDegree(); j++) {
      secondaryLocalApproximationSpace2d.push_back(
                new OnceDerivableTensorProductFunction2D<const OnceDerivableFunction>(
                   new MonomialFunction(i),
                   new MonomialFunction(j)));
   }
}
```

Listing 3.4 Beispielproblemdefinition im Prototyp 1 für das Driven-Cavity-Problem: Definition der Cover für Geschwindigkeit und Druck

```
Domain2D saddlePointDomain2d(Interval(-1, 1), Interval(-1, 1));

Cover2D &saddlePointPrimaryCover2d = RegularCover2D(saddlePointDomain2d, 4, 4, 1.5);
Cover2D &saddlePointSecondaryCover2d = RegularCover2D(saddlePointDomain2d, 3, 3, 1.5);

for (size_t i = 0; i < saddlePointPrimaryCover2d.m_patches.size(); i++) {
    for (size_t j = 0; j < localApproximationSpace.size(); j++) {
        saddlePointPrimaryCover2d.m_patches[i].m_firstDimension
            .addLocalApproximationFunction(new
                TransformableFunction(localApproximationSpace[j]));
        saddlePointPrimaryCover2d.m_patches[i].m_secondDimension
            .addLocalApproximationFunction(new
                TransformableFunction(localApproximationSpace[j]));
    }
}

for (size_t i = 0; i < saddlePointSecondaryCover2d.m_patches.size(); i++) {
    for (size_t j = 0; j < localApproximationSpace.size(); j++) {
        saddlePointSecondaryCover2d.m_patches[i].m_firstDimension
            .addLocalApproximationFunction(new
                TransformableFunction(localApproximationSpace[j]));
        saddlePointSecondaryCover2d.m_patches[i].m_secondDimension
            .addLocalApproximationFunction(new
                TransformableFunction(localApproximationSpace[j]));
    }
}
```

Listing 3.5 Beispielproblemdefinition im Prototyp 2 für das Driven-Cavity-Problem: Definition der Cover für Geschwindigkeit und Druck

```
Domain2D domain(-1, 1, -1, 1);

RegularCover2D primaryCover(domain,
    Settings::currentSettings()->primaryCoverNumPatchesFirstDimension(),
    Settings::currentSettings()->primaryCoverNumPatchesSecondDimension(),
    Settings::currentSettings()->primaryCoverStretchFirstDimension(),
    Settings::currentSettings()->primaryCoverStretchSecondDimension(),
    localApproximationSpace2d);
RegularCover2D secondaryCover(domain,
    Settings::currentSettings()->secondaryCoverNumPatchesFirstDimension(),
    Settings::currentSettings()->secondaryCoverNumPatchesSecondDimension(),
    Settings::currentSettings()->secondaryCoverStretchFirstDimension(),
    Settings::currentSettings()->secondaryCoverStretchSecondDimension(),
    secondaryLocalApproximationSpace2d);

SaddlePointCover2D cover(domain, primaryCover, secondaryCover);
```

Listing 3.6 Beispielproblemdefinition im Prototyp 1 für das Driven-Cavity-Problem: Definition des Problem-Objekts

```
std::vector<double> saddlePointProblemCoefficients2d(1, 0);
saddlePointProblemCoefficients2d[0] = 0.0;

std::vector<double> saddlePointBoundaryCoefficients2d(1, 0);
saddlePointBoundaryCoefficients2d[0] = 1.0;

StokesSaddlePointProblem2D saddlePointProblem2d(
          new GaussQuadratureRule2D(),
          new GaussQuadratureRule(),
          new VectorValuedFunction2D(
              new TensorProductFunction2D(new
                  PolynomialFunction(saddlePointProblemCoefficients2d), new
                  PolynomialFunction(saddlePointProblemCoefficients2d)),
              new TensorProductFunction2D(new
                  PolynomialFunction(saddlePointProblemCoefficients2d), new
                  PolynomialFunction(saddlePointProblemCoefficients2d))),
          saddlePointPrimaryCover2d,
          saddlePointSecondaryCover2d,
          new VectorValuedFunction(ZeroFunction::s_instance, ZeroFunction::s_instance),
          new VectorValuedFunction(new
              PolynomialFunction(saddlePointBoundaryCoefficients2d),
              ZeroFunction::s_instance),
          new VectorValuedFunction(ZeroFunction::s_instance, ZeroFunction::s_instance),
          new VectorValuedFunction(ZeroFunction::s_instance, ZeroFunction::s_instance));
```

Listing 3.7 Beispielproblemdefinition im Prototyp 2 für das Driven-Cavity-Problem: Definition des Problem-Objekts

```
std::vector<Function*> boundaryValuesU;
std::vector<Function*> boundaryValuesV;
for (std::list<BoundarySegment2D>::const_iterator i = domain.boundarySegments().begin(); i
    != domain.boundarySegments().end(); i++) {
   boundaryValuesV.push_back(new ConstFunction(0));
   if ((*i).orientation() == BoundarySegment2D::NegativeX) {
      boundaryValuesU.push_back(new ConstFunction(1));
   } else {
      boundaryValuesU.push_back(new ConstFunction(0));
   }
}

PumSaddlePointProblem2D problem(
          cover,
          new GaussQuadratureRule2D(),
          new GaussQuadratureRule(),
          new PoissonBilinearForm2D(),
          new PoissonLinearForm2D(new ConstFunction2D(0), boundaryValuesU),
          new PoissonLinearForm2D(new ConstFunction2D(0), boundaryValuesV),
          new StokesSecondaryUBilinearForm2D(),
          new StokesSecondaryVBilinearForm2D(),
          new ZeroLinearForm2D());
```

Die Listings 3.6 und 3.7 zeigen die eigentliche Definition der Partition of Unity Probleme. In Prototyp 1 war es noch nötig mit vektorwertigen Funktionen zu hantieren, dies wurde durch die stärkere Orientierung an den Bedürfnissen für das Stokes-Sattelpunktproblem in Prototyp 2 überflüssig. In Prototyp 1 müssen die Randwerte für jede Seite des rechteckigen Gebiets explizit pro Seite angegeben werden. In Prototyp 2 erfolgt die Definition von Randwerten für alle Randstücke anhand deren Orientierung und ist daher für allgemeinere Ränder verwendbar. Ebenso sind die Randwerte in Prototyp 2 direkt in die Linearform integriert, welche in Prototyp 1 noch implizit in der Problemdefinition enthalten ist.

3.6.2 Leistungsverhalten

Zur Messung der Geschwindigkeit und des Speicherverbrauchs werden zweidimensionale Probleme verwendet, weil eindimensionale Probleme nicht ausreichend Auslastung erzeugen können und dadurch die Ergebnisse verfälschen. Es wird jeweils ein Poisson-Problem und ein Stokes-Sattelpunktproblem mit verschiedene Covers und Ansatzräumen gelöst. Zur Zeitmessung kommt die C++-Funktion `clock()` zum Einsatz. Daher sind alle Zeiten in Tabelle 3.1 in `clock()`-Einheiten angegeben. Auf dem verwendeten Testsystem ist eine `clock()`-Einheit äquivalent zu einer Millisekunde. Da der verwendete Code nicht parallelisiert ist, wird nur die Rechenzeit mit einem Thread gemessen. Für die Messung werden alle Ausgaben von Matrizen, Vektoren und Lösungen abgeschaltet, um nur die reine Rechenzeit zu messen. In Prototyp 2 wird als Regularisierungsparameter 100 fest eingestellt, weil die Berechnung für die große der Cover zu lange dauert und der gewählte Parameter ausreichend genau ist, um das Ergebnis nicht signifikant zu verfälschen. Ein Messdurchlauf besteht darin, dass das Programm zunächst drei mal ohne Messung ausgeführt wird, um die Effekte von Caching durch das Betriebssystem zu minimieren und anschließend fünf mal mit Messung. In der Tabelle wird jeweils das Minimum, das Maximum und der Durchschnitt der gemessenen Werte angegeben. In Prototyp 2 lassen sich der Aufwand für die Vorauswertung und der Aufwand für die Integration nicht getrennt messen, daher wird hier für beide Schritte nur ein Messwert angegeben. Für den Prototyp 2 gibt es keine Messungen zum Driven-Cavity-Stokes-Problem, weil der verwendete Uzawa-Algorithmus zu langsam konvergiert.

Die Messungen werden für folgende Probleme durchgeführt:

Poisson-Problem:

$$\begin{aligned} \Delta u &= -1 \quad \text{auf } \Omega \\ u &= 1 \quad \text{auf } \partial\Omega \end{aligned} \tag{3.22}$$

mit $\Omega = [-1,1] \times [-1,1]$.

		Prototyp 1				Prototyp 2		
Cover		Vorauswertung	Integration	Lösung	Speicher (KByte)	Vorauswertung und Integration	Lösung	Speicher (KByte)
	Durchschnitt	143	123	397		1141	60	
10x10,p=2	Minimum	134	121	384	20728	1129	59	3552
	Maximum	148	125	408		1155	62	
	Durchschnitt	174	167	1070		1386	106	
11x11,p=2	Minimum	165	163	1039	27068	1271	105	3884
	Maximum	191	175	1110		1440	108	
	Durchschnitt	211	219	2143		1662	125	
12x12,p=2	Minimum	198	215	2106	34268	1633	118	5124
	Maximum	231	223	2197		1694	145	
	Durchschnitt	238	283	3812		1952	187	
13x13,p=2	Minimum	213	280	3798	42672	1939	185	5700
	Maximum	252	285	3827		1975	190	
	Durchschnitt	287	360	5990		2249	205	
14x14,p=2	Minimum	275	354	5952	53228	2212	204	6260
	Maximum	306	366	6050		2298	207	
	Durchschnitt	221	386	4326		3582	741	
10x10,p=3	Minimum	210	381	4266	40304	3472	739	6800
	Maximum	239	396	4343		3647	747	

Tabelle 3.1: Laufzeit der einzelnen Berechnungsschritte in beiden Prototypen für das Poisson-Problem. Für den Speicherverbrauch ist nur ein Wert angegeben, da dieser über alle Läufe konstant war. Die Angabe 10x10,p=2 zum Cover ist wie folgt zu lesen: 10 mal 10 Patches mit einem lokalen Ansatz Raum bis zu einem Polynomgrad 2 pro Dimension.

Stokes-Sattelpunktproblem (Driven Cavity):

$$
\begin{aligned}
-\Delta u + \nabla p &= 0 && \text{auf } \Omega \\
\operatorname{div} u &= 0 && \text{auf } \Omega \\
u &= 0 && \text{auf } \partial\Omega \setminus \Gamma \\
u &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} && \text{auf } \Gamma
\end{aligned}
\tag{3.23}
$$

mit $\Omega = [-1,1] \times [-1,1]$ und $\Gamma = [-1,1] \times [1]$.

3.6.3 Erweiterbarkeit

Die Erweiterbarkeit war insbesondere für den Prototyp 2 eine wichtige Anforderung. Beide Prototypen sind so angelegt, dass die lokalen Ansatzräume ausgetauscht werden können. In Prototyp 1 lassen sich Enrichments mit der Methode `addLocalApproximationFunction` einfach einzelnen Patches zuweisen. In Prototyp 2 lassen sich Enrichments nur global in alle lokalen Ansatzräume einfügen. In beiden Prototypen können beliebige zweidimensionale Funktionen als lokale Ansatzfunktionen verwendet werden.

Als Domain sind in Prototyp 1 nur rechteckige Gebiete zugelassen. Prototyp 2 ist für Gebiete, die aus einer Menge Tensorproduktintervallen bestehen, vorbereitet. Allgemeinere Gebiete sind bisher nicht vorgesehen, da dafür eine Approximation der Gebiete mit Dreiecken implementiert werden müsste.

Prototyp 1 kann, aufgrund des Algorithmus zur Berechnung für Integrationszellen, nur mit regulären Covern umgehen. Prototyp 2 dagegen kann auch mit Covern aus einer beliebigen Menge rechteckiger Patches umgehen. Diese müssen allerdings programmatisch erstellt werden.

Die Lösung bisher nicht implementierter Probleme, erfordert in Prototyp 1 die Implementierung einer eigenen Version des Algorithmus. In Prototyp 2 muss nur der passende Algorithmus mit einer anderen Bilinearform und Linearform instantiiert werden. Dadurch ist es in Prototyp 2 wesentlich einfacher neue Probleme zu implementieren.

In beiden Prototypen ist es möglich die Quadraturregeln auszutauschen. Dazu muss lediglich ein Interface implementiert werden. Weiterhin besteht damit die Möglichkeit die Quadraturregeln adaptiv an die für ein Integral benötigte Genauigkeit anzupassen.

4 Fazit

Die beiden Prototypen zeigen, wie eine Implementierung der Partition of Unity Methode für Sattelpunktprobleme aussehen kann. Um die hier vorgestellten Lösungsansätze für praktische Anwendungen einsetzen zu können, sind allerdings noch einige Schritte nötig. Insbesondere müsste die Performance erheblich verbessert werden, wie beispielsweise in [Sch05] beschrieben. Die beiden Prototypen dienen aber vorrangig als experimentelle Plattform für Optimierungen und Erweiterungen, sodass auf Grundlage eines einfach verständlichen Quellcodes alternative Implementierungen entwickelt werden können.

Während der Entwicklung der beiden Prototypen hat sich gezeigt, dass es sinnvoll ist, zunächst den gewünschten Algorithmus iterativ bis zum vollen Funktionsumfang auszubauen und dann anschließend mit den sich daraus ergebenden Anforderungen eine neue Implementierung samt neuer Architektur zu erstellen. In Prototyp 1 hat sich gezeigt, dass eine immer weitere Optimierung des Algorithmus zu einer Zersetzung der Architektur führt und dadurch nachfolgende Optimierungen immer mehr Aufwand kosten. Dieser Effekt tritt umso stärker auf, je mehr Algorithmenschritte durch eine Optimierung verändert werden.

Für den Prototyp 2 waren ursprünglich mehr Funktionen geplant, konnten jedoch aus Zeitgründen nicht realisiert werden. Da die Planung bereits erfolgt ist, bietet sich ein Einstieg in die Weiterentwicklung mit den folgenden Funktionen an: die Ausnutzung der Tensorproduktstruktur, Implementierung eine Referenzintegration, Vervollständigung der Unterstützung komplexerer Gebiete, die Verwendung von Enrichments und Unterstützung der Vorkonditionierung der Matrix. Weiterhin kann als nächster Schritt die Implementierung der Sattelpunktprobleme, unter Verwendung der in dieser Diplomarbeit gewonnenen Erkenntnisse, in CraSS angegangen werden.

Literaturverzeichnis

[Bra07] D. Braess. *Finite Elemente - Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer-Verlag Berlin Heidelberg, 2007. (Zitiert auf den Seiten 13, 23, 24 und 44)

[Fag10] O. A. Fagerlund. Multi-core programming with OpenCL: performance and portability : OpenCL in a memory bound scenario, 2010. (Zitiert auf Seite 50)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (Zitiert auf Seite 31)

[Lei12] J. Leibinger. Fluidsimulationen mit der Partition of Unity Methode, 2012. (Zitiert auf Seite 29)

[LL07] J. Ludewig, H. Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007. (Zitiert auf Seite 29)

[MB96] J. Melenk, I. Babuška. The Partition of Unity Finite Element Method: Basic Theory and Applications, 1996. (Zitiert auf Seite 15)

[Nit71] J. Nitsche. Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, Band 36, S. 9–15. Springer Berlin / Heidelberg, 1971. (Zitiert auf Seite 21)

[par] ParaView. URL http://www.paraview.org/. (Zitiert auf Seite 29)

[qts] Qt C++-Styleguide. URL http://wiki.qt-project.org/Coding_Style. (Zitiert auf Seite 30)

[Sch05] M. A. Schweitzer. Efficient Implementation and Parallelization of Meshfree and Particle Methods - The Parallel Multilevel Partition of Unity Method, 2005. (Zitiert auf den Seiten 22, 29, 30 und 65)

[She68] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, S. 517–524. ACM, New York, NY, USA, 1968. doi:10.1145/800186.810616. URL http://doi.acm.org/10.1145/800186.810616. (Zitiert auf Seite 16)

Alle URLs wurden zuletzt am 06.06.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Christian Dittrich)