FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Patch-adaptive Relaxation als Glätter im Mehrgitterverfahren

Iris Christadler

Studienarbeit

Patch-adaptive Relaxation als Glätter im Mehrgitterverfahren

Iris Christadler

Studienarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. M. Kowarschik
Bearbeitungszeitraum:	1.4.2003 – 31.12.2003

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 22. Januar 2004 ....

Inhaltsverzeichnis

Zusammenfassung

Mehrgitterverfahren haben sich zur Lösung linearer Gleichungssysteme aus der Diskretisierung partieller Differentialgleichungen bewährt. Ein wichtiger Teil dieser Verfahren ist – abgesehen von Interpolation und Restriktion – der verwendete Glätter. Bei komplizierteren Gleichungssystemen ist die Konvergenz des Verfahrens stark von diesem abhängig. Um einen Glätter noch effizienter zu machen und damit die Konvergenzrate des Mehrgitterverfahrens zu verbessern, gibt es den Ansatz eines adaptiven Glätters. Dieser relaxiert nacheinander die Stellen mit dem schlechtesten Residuum, um so der Lösung möglichst schnell näherzukommen. Die grundlegende Idee zu einem solchen Glätter findet sich in [Rü93]. Eine Implementierung dazu lieferte bereits die Studienarbeit von V. Daum [Dau01]. In der vorliegenden Arbeit wurde die Idee der adaptiven Relaxation nun auf Patches erweitert. Der dadurch entstandene Glätter kann selbstständig die Patches bestimmen, die weitere Glättungen benötigen. Die Erweiterung auf Patches hat den Vorteil, daß dadurch die Cachehierarchie eines Computers optimal ausgenutzt werden kann. So erwartet man von einem solchen adaptiven Glätter nicht nur eine bessere Konvergenzrate sondern gleichzeitig durch die Cacheoptimierung auch geringere Laufzeiten. Das erste Ziel, die Konvergenzrate mithilfe des Glätters zu verbessern, konnte in dieser Arbeit exemplarisch an dem Modellproblem des L-Gebiets gezeigt werden. Die Frage, inwieweit die Cacheperformance den Mehraufwand, den ein adaptiver Glätter benötigt, ausgleichen kann, bleibt folgenden Arbeiten vorbehalten. Einfache Zeitmessungen an dem vorliegenden Programm lassen dies vermuten. Eingehende Tests mit Profiling-Tools bieten sich an der momentanen Fassung des Programmes allerdings nicht an, da während der Implementierung nicht darauf geachtet wurde diverse Optimierungsmöglichkeiten auszunutzen. Viel eher ist das vorliegende Programm, – welches sich auf der beigelegten CD befindet – als ein Framework zu verstehen, bei dem viele Möglichkeiten offengehalten wurden. So beinhaltet es unter anderem die Möglichkeit, verschiedene Mehrgitterverfahren mit unterschiedlichen Glättern zu kombinieren, zu testen und zu vergleichen.

Kapitel 1

Vorwort

Die vorliegende Arbeit versucht, den Entstehungsprozeß des beigefügten Programmes abzubilden. Da ein Patch-adaptiver Glätter ein Teilproblem bei der Konzeptionierung von Mehrgitterverfahren ist, wird in den folgenden Kapiteln davon ausgegangen, daß der Leser ein gewisses Grundwissen sowohl über Mehrgitterverfahren als auch über iterative Verfahren zur Lösung linearer Gleichungssysteme im allgemeinen und dem Verfahren von Gauß-Seidel im speziellen besitzt. In den Kapiteln 2 bis 4 wird versucht, den Weg, der zur Entstehung des Patch-adaptiven Glätters geführt hat, auch in der Theorie zu beschreiten.

Ein interessierter Leser, der einen Einstieg in die Arbeits- und Wirkungsweise von Mehrgitterverfahren sucht, sei auf [B$^+$00] verwiesen. Dieses Buch beinhaltet alle Grundlagen, um die hier vorgestellte Theorie nachvollziehen zu können. Genaue mathematische Abschätzungen, Beweise und ähnliches, die über das Verfahren der punktweise adaptiven Relaxation (Abschnitt 3.1) gemacht werden können, sind nachzulesen in [Rü93]. Als tiefergreifender Einstieg in die Mehrgittertheorie bietet sich [TOS01] an.

Wie oben erwähnt, versucht sich der Aufbau dieser Arbeit an die Entwicklung unseres Programmes anzulehnen.

Kapitel 2 bietet eine sehr kurze Einführung in lineare Gleichungssysteme, die aus der Diskretisierung von differentiellen Partialgleichungen stammen, und Verfahren zur Lösung derselbigen. Die darin vorgestellten iterativen Verfahren können, außer als Löser für lineare Gleichungssysteme, auch als Glätter im Mehrgitterverfahren eingesetzt werden. In diesem Kapitel wird eine erste Version eines adaptiven Glätters, nämlich das Gauß-Southwell-Verfahren, vorgestellt.

Kapitel 3 beschäftigt sich im Anschluß daran mit dem in [Rü93] vorgestellten Verfahren zur adaptiven Relaxation. Dieses ist eine Weiterentwicklung des Gauß-Southwell-Verfahrens und gleichzeitig die Grundlage für den im Zuge dieser Arbeit entstandenen Patch-adaptiven Glätter. Mit diesem Verfahren haben sich bereits andere Studien- beziehungsweise Diplomarbeiten beschäftigt, deren Ergebnisse starken Einfluß auf die Entwicklung und Implementierung der patchweisen Methode hatten. Deshalb schließt sich an die Vorstellung der punktweise adaptiven Relaxation eine kurze Zusammenfassung der bisher gewonnen Ergebnisse an.

Kapitel 4 ist komplett der Theorie hinter der patchweise adaptiven Relaxation gewidmet. Hier wird in einzelnen Schritten gezeigt, wie der punktweise Fall für Patches abgewandelt wurde und welche Vorteile durch die Verwendung von Patches zu erwarten sind. Es wurde versucht, möglichst klar zwischen der Theorie, die für die Entwicklung des Algorithmus vonnöten war, und den programmiertechnischen Designentscheidungen der Implementierung zu trennen.

Letztere werden in **Kapitel 5** vorgestellt. Hier wird vor allem auf die einzelnen implementierten Klassen und deren Funktion eingegangen. Der gesammte Programmcode ist mit Kommentaren für eine Dokumentation mittels Doxygen versehen, welche im HTML-Format und als Manpages zur Verfügung stehen. Diese auf der CD befindliche Dokumentation ergänzt Kapitel 5 um weitere programmiertechnische Details.

Kapitel 6 beleuchtet nun abschließend die in den Tests gewonnen Resultate, einschließlich der erfreulichen Bestätigung unserer Hypothese, mithilfe des Patch-adaptiven Glätters eine verbesserte Konvergenz zusammen mit einer geringeren Anzahl notwendiger Relaxationen zu erreichen. Da das Programm, das während dieser Studienarbeit entstand, nun zu einer Art Framework angewachsen ist, wird erwartet, daß sich noch weitere positive Ergebnisse – auch im Hinblick auf Cacheperformanz und Laufzeit – in sich der Arbeit anschließenden Tests erzielen lassen.

Kapitel 2

Von Gauß-Seidel über Gauß-Southwell hin zum Punkt-adaptiven Löser

2.1 Lineare Gleichungssysteme aus partiellen Differentialgleichungen

Ein lineares Gleichungssystem (LGS) läßt sich schreiben als

$$Au^* = f, \tag{2.1}$$

mit der Matrix A, einer gegebenen rechten Seite f und dem gesuchten Vektor u^*. Stammt dieses Gleichungssystem aus der Diskretisierung einer partiellen Differentialgleichung, ist die Matrix A meistens *dünn* besetzt.

Bei den in dieser Studienarbeit verwendeten Modellproblemen ist die Matrix symmetrisch und positiv definit. Der zu ihrer Lösung verwendete Stern besteht aus höchstens neun Einträgen, und zwar jeweils einem Eintrag für:

NW (north-west)	N (north)	NE (north-east)
W (west)	C (center)	E (east)
SW (south-west)	S (south)	SE (south-east)

(siehe auch Abbildung 2.2)

Eine genauere Beschreibung der verwendeten Modellprobleme findet sich in Kapitel 5.
Wie in Abbildung 2.3 zu sehen ist, besitzt die Matrix A eine Block-Tridiagonal-Struktur mit insgesamt neun Bändern.

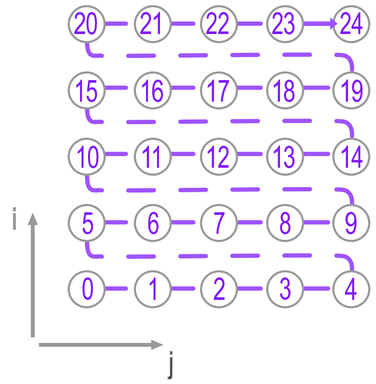


Abbildung 2.1: Anordnung des Vektors in einem Gitter.

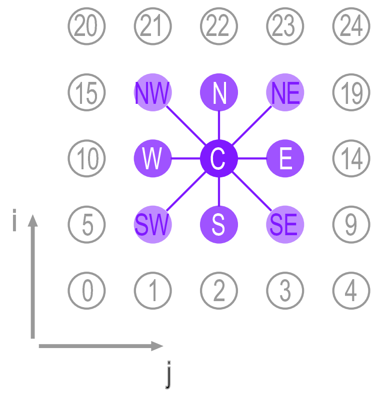


Abbildung 2.2: 9-Punkt-Stern.

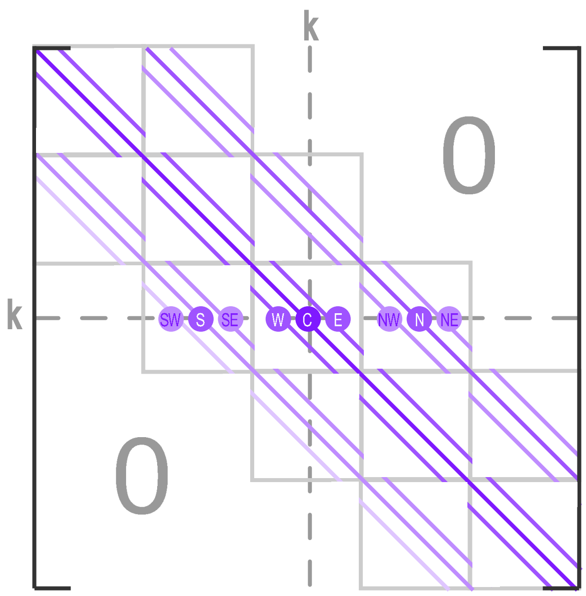


Abbildung 2.3: Matrix mit 9 Bändern in Block-Tridiagonal-Struktur.

2.2 Iterative Löser für lineare Gleichungssysteme

Um Gleichungssysteme dieser Art zu lösen, gibt es zwei Möglichkeiten:

- Direkte Lösungsmethoden,
- Iterative Verfahren.

Direkte Lösungsmethoden (wie zum Beispiel die LU-Zerlegung) sind häufig genutzte Methoden. Eine Vielzahl direkter Löser steht zum Beispiel im Rahmen des LAPACK-Pakets dem Benutzer zur Verfügung. Diese Methoden besitzen nicht die optimale Komplexität, sind dafür aber sehr robust. So kann ein direkter Löser für eine Vielzahl unterschiedlicher Gleichungssysteme eingesetzt werden.

Im Gegensatz dazu stellen *iterative Verfahren* oft hohe Anforderungen an die Eigenschaften des linearen Gleichungssystems. Dieses muß zum Beispiel – im Fall von Mehrgitterverfahren – symmetrisch und positiv definit sein um die Konvergenz des Verfahrens sicherzustellen. Da iterative Verfahren nur für spezielle Probleme geeignet sind, können sie jedoch auch die Vorteile dieser Probleme ausnutzen (zum Beispiel die dünn besetzte Matrix). So benötigen sie nicht nur weniger Speicherplatz sondern auch geringeren Rechenaufwand als ein direkter Löser.

Iterative Verfahren laufen dabei nach folgendem Schema ab: Man nähert sich, ausgehend von einer Startnäherung $u^{(0)}$, in mehreren Iterationen $u^{(1)}, u^{(2)}, \ldots$ immer weiter an die exakte Lösung des Gleichungssystems u^* an. Die Iterierte $u^{(k)}$ berechnet sich hierbei lediglich aus vorherigen Iterierten $u^{(i)}$ mit $i < k$; oft sogar nur aus $u^{(k-1)}$.

2.2.1 Gauß-Seidel

Algorithmus 2.1 Gauß-Seidel

repeat
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      `relax` $(u_{(i,j)})$
    end for
  end for
until convergence

Einer der wohl bekanntesten iterativen Löser für lineare Gleichungssysteme ist das Gauß-Seidel-Verfahren (GS-Verfahren). Mathematisch läßt sich der Iterationsschritt folgendermaßen ausdrücken

$$u_i^{(k+1)} = \frac{1}{a_{ii}} \cdot (f_i - \sum_{j=1}^{i-1} a_{ij} u_j^{(k+1)} - \sum_{j=i+1}^{(N-1)} a_{ij} u_j^{(k)}). \tag{2.2}$$

Handelt es sich bei A um eine $\mathbf{R}^{m \times m}$-Matrix und ordnet man die Vektoren u und f – beide aus $\mathbf{R}^m$ – in einem 2D-Gitter an, kann der Pseudocode für das Verfahren wie in Algorithmus 2.1 implementiert werden. Das Lösungsgitter u besteht dabei – wie auch f – aus $n \times n$ Unbekannten, wobei $n \cdot n = m$ gilt. Da im Programmcode nur mit gitterförmig angeordneten Vektoren gearbeitet wird, bezeichnet $u_{(i,j)}$ ab sofort die Komponente des Vektors u, die auf den Koordinaten (i, j) des Gitters liegt. Dabei gilt: $u_{(i,j)} = u_{i \cdot \texttt{size}+j}$. (Vergleiche auch Abbildung 2.1.)

Schränkt man das Verfahren auf 9-Punkt-Sterne ein, so vereinfacht sich `relax` $(u_{(i,j)})$ zu

$$\begin{array}{lllllllll} u_{(i,j)} & \leftarrow & \frac{1}{a_C} \cdot (f_C - (& a_{SW} \cdot u_{SW} & + & a_S \cdot u_S & + & a_{SE} \cdot u_{SE} & + \\ & & & a_E \cdot u_E & + & a_{NE} \cdot u_{NE} & + & a_N \cdot u_N & + \\ & & & a_{NW} \cdot u_{NW} & + & a_W \cdot u_W & & &)). \end{array}$$

2.2.2 Gauß-Southwell

Die Idee hinter diesem iterativen Verfahren ist, immer den Punkt mit der größten Komponente des Residuum zu glätten, um sich damit effizienter der gesuchten Lösung u^* zu nähern. (Zum Ablauf des Verfahrens siehe Algorithmus 2.2.)

Algorithmus 2.2 Gauß-Southwell

repeat
  `computeResidual()`
  `pick`(i, j) `where` $r_{(i,j)} > r_{(x,y)}$; $x, y \in 1, 2, \ldots, n$
  `relax` $(u_{(i,j)})$
until convergence

Nimmt man die Anzahl der Aufrufe von `relax()` als Maß für die Effizienz, kann dieses Ziel erreicht werden. Jedoch verursacht die Berechnung des Residuums (`computeResidual()`) vor jeder Punktrelaxation einen enormen rechnerischen Overhead, der dazu führt, daß das Verfahren im allgemeinen deutlich längere Laufzeiten als ein Standard-GS hat. Eine einfache Erklärung dazu liefert der Pseudocode der Funktion `computeResidual()` (siehe Algorithmus 2.3).

Algorithmus 2.3 `computeResidual()`

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $r_{(i,j)} \leftarrow$ `computeResidualAt` $(u_{(i,j)})$
  end for
end for

mit

$$\begin{array}{lllllllll} \bar{r}_{(i,j)} & \leftarrow & \frac{1}{a_C} \cdot (f_C - (& a_{SW} \cdot u_{SW} & + & a_S \cdot u_S & + & a_{SE} \cdot u_{SE} & + \\ & & & a_E \cdot u_E & + & a_{NE} \cdot u_{NE} & + & a_N \cdot u_N & + \\ & & & a_{NW} \cdot u_{NW} & + & a_W \cdot u_W & + & a_C \cdot u_C &)). \end{array}$$

Beachtet man dazu noch, daß die Berechnung von `computeResidualAt()` in der Anzahl der Gleitpunktoperationen genauso teuer ist wie die Berechnung von `relax()`, wird klar, daß eine Relaxation im Gauß-Southwell-Verfahren so teuer ist wie die Relaxation des gesamten Gitters während Gauß-Seidel. Auch die verminderte Anzahl von Punktrelaxationen kann diesen Nachteil nicht aufwiegen.

Interessant wird dieses Verfahren aber, wenn man nicht jedesmal das gesamte Residuum bestimmt, sondern lediglich die Komponenten neuberechnet, die sich durch die Relaxation geändert haben. Allerdings benötigt man zusätzlich eine effiziente Strategie, um auch die Auswahl der größten Komponenten (`pick()`) nicht zu teuer werden zu lassen.

Ein sinnvoll abgewandelter Algorithmus, der die beiden oben beschriebenen Probleme zumindest bis zu einem gewissen Grad löst, ist die Punkt-adaptive Relaxation nach U. Rüde. Dieses Verfahren bildet die Grundlage der vorliegenden Studienarbeit und soll im nächsten Kapitel genauer erläutert werden.

Kapitel 3

Vom Punkt zum Patch

3.1 U. Rüdes punktweise adaptive Relaxation

Der dieser Arbeit zugrunde liegende Algorithmus von U. Rüde ist zu finden in [Rü93].

Eine auf das wesentliche reduzierte Form des Verfahrens findet sich als Pseudocode unter Algorithmus 3.1. Die Erklärung der einzelnen Teilschritte und eine Herleitung der durch den Algorithmus nach seiner Terminierung definierten Schranke für die Maximumsnorm des skalierten Residuums von u findet sich im Anschluß an die notwendigen Definitionen.

3.1.1 Der Pseudocode

Algorithmus 3.1 punktweise adaptiver Gauß-Seidel

Require: tolerance τ, initial Guess $u^{(0)}$, activeSet $\tilde{S}(\tau, u)$

$u \leftarrow u^{(0)}$
while $\tilde{S}(\tau, u) \neq \emptyset$ **do**
    pick point $u_{(i,j)} \in \tilde{S}(\tau, u)$
    delete $u_{(i,j)}$ from $\tilde{S}(\tau, u)$
    $\bar{r}_{(i,j)} \leftarrow$ `computeScaledResidual` $(u_{(i,j)})$
    if $\bar{r}_{(i,j)} > \tau$ **then**
        `relax` $(u_{(i,j)})$
        `putNeighboursInActiveSet` $(\tilde{S}(\tau, u), u_{(i,j)})$
    end if
end while

3.1.2 Definitionen

Residuum (Definition)

Das Residuum $r^{(k)}$ des Vektors $u^{(k)}$ berechnet sich als

$$r^{(k)} := f - Au^{(k)}.$$

Skaliertes Residuum (Definition)

Das skalierte Residuum $\bar{r}^{(k)}$ des Vektors $u^{(k)}$ ist

$$\bar{r}^{(k)} := D^{-1}(f - Au^{(k)}),$$

mit $D = \mathrm{diag}(A)$.

Das skalierte Residuum $\bar{r}^{(k)}_{(i,j)}$ für einen Punkt $u_{(i,j)}$ ist

$$\bar{r}^{(k)}_{(i,j)} := e^T_{(i,j)}\bar{r},$$

wobei $e_{(i,j)}$ den Einheitsvektor an Stelle (i, j) bezeichnet.

Beachte: Hat man das skalierte Residuum $\bar{r}^{(k)}_{(i,j)}$ berechnet, vereinfacht sich `relax` $(u_{(i,j)})$ zu

$$u^{(k+1)}_{(i,j)} \leftarrow u^{(k)}_{(i,j)} + \bar{r}^{(k)}_{(i,j)}.$$

Fehler (Definition)

Der Fehler $e^{(k)}$ einer Lösung $u^{(k)}$ berechnet sich aus

$$e^{(k)} := u^* - u^{(k)}.$$

Somit ergibt sich für das Verhältnis von Residuum und Fehler folgende Aussage

$$Ae^{(k)} = A(u^* - u^{(k)}) = Au^* - Au^{(k)} = f - Au^{(k)} = r^{(k)}. \tag{3.1}$$

Bemerkung: Für Mehrgitterverfahren bietet das skalierte Residuum gegenüber dem Residuum einen großen Vorteil. In diesen Verfahren ist die Matrix A abhängig vom jeweiligen Level, da sie während der Diskretisierung der partiellen Differentialgleichung mit der Gitterweite $h = 1/$`size` skaliert wird. Betrachtet man als Abbruchkriterium des Glätters den Fehler, ist dieser unabhängig von der Gitterweite h. Da die analytische Lösung u^* im allgemeinen aber nicht zur Verfügung steht, verwendet man als Abbruchkriterium häufig das Residuum. Da dieses aber – im Gegensatz zum skalierten Residuum – von der Gitterweite abhängig ist, verwendet man im Mehrgitterfall besser das von h unabhängige (siehe dazu Gleichung 3.1) skalierte Residuum.

Strikt aktive Menge (Definition)

Die strikt aktive Menge $S(\tau, u^{(k)})$ ist definiert als

$$S(\tau, u^{(k)}) := \{(i, j)|\, |\bar{r}^{(k)}_{(i,j)}| \geq \tau\} \quad \text{mit } \tau > 0.$$

Aktive Menge (Definition)

Eine aktive Menge $\tilde{S}(\tau, u^{(k)})$ ist

$$S(\tau, u^{(k)}) \subseteq \tilde{S}(\tau, u^{(k)}) \subseteq \{(0, 0), (0, 1), \ldots, (n, n)\}.$$

Beachte: Auch die Menge aller Zweiertupel (i, j) ist eine mögliche aktive Menge.

Nachbarschaft (Definition)

Die Nachbarschaft $\text{Conn}_{(i,j)}$ eines Punktes $u_{(i,j)}$ ist definiert als

$$\text{Conn}_{(i,j)} := \{(k,l)|(k,l) \neq (i,j) \wedge a_{(i,j),(k,l)} \neq 0\}$$

Bemerkung: Die Nachbarschaft ist somit die Menge aller Punkte, auf die von Punkt $u_{(i,j)}$ mithilfe seines Sterns zugegriffen wird. Dabei werden nur die Nachbarpunkte betrachten, für die der Eintrag im Stern $\neq 0$ ist. $\bar{C}$ bezeichnet die maximale Anzahl von Nachbarn für ein Gleichungssystem $Au = f$. Somit gilt im Fall von 9-Punkt-Sternen $\bar{C} = 9$.

Maximumsnorm (Definition)

Die Maximumsnorm $\|.\|_\infty$ eines Vektors $u^{(k)}$ ist definiert als:

$$\|u^{(k)}\|_\infty := \max_{(i,j)=(0,0),(0,1)\dots,(n,n)} |u_{(i,j)}|.$$

3.1.3 Die Vorgehensweise des Algorithmus

Mit den gegebenen Definitionen läßt sich Algorithmus 3.1 nun wie folgt beschreiben:
Der Algorithmus benötigt außer der Startlösung $u^{(0)}$ eine festgelegte Toleranz τ. Diese Toleranz bestimmt die Güte von $u^{(k)}$, nachdem der Algorithmus terminiert ist. Die strikt aktive Menge $\tilde{S}$ ist abhängig von der gewählten Toleranz. Eine aktive Menge beinhaltet mindestens alle Punkte, deren skaliertes Residuum über der Toleranz τ liegen. Auf dieser aktiven Menge wird so lange gearbeitet, bis sie keine Elemente mehr enthält. Ist die aktive Menge leer, gibt es keinen Punkt $u_{(i,j)}$, dessen skaliertes Residuum größer als τ ist. Somit gilt für die Maximumsnorm des skalierten Residuums nach Beendigung des Algorithmus $\|u^{(k)}\|_\infty \leq \tau$.

Nun muß noch geklärt werden, warum bei jedem Teilschritt des Verfahrens die aktive Menge weiterhin alle Punkte enthält, deren skaliertes Residuum über der Toleranz τ liegt. Für die aktive Menge beim Start des Verfahrens kann dies gefordert werden. Eine Möglichkeit, die aktive Menge zu bestimmen, beinhaltet, das skalierte Residuum für jeden Punkt zu berechnen und somit die strikt aktive Menge S zu bestimmen. Weniger Aufwand bereitet eine zweite aktive Menge, welche einfach alle Punkte $u_{(i,j)}$ enthält und alternativ als Startmenge gewählt werden kann. Diese ist eine Obermenge der strikt aktiven Menge und damit per Definition eine aktive Menge.

Die Auswahl eines Punktes aus der aktiven Menge in Zeile 2 verändert die aktive Menge nicht. Erst das Löschen des Punktes eine Zeile darauf löscht eventuell einen Punkt aus der Menge, dessen skaliertes Residuum größer als τ ist. Allerdings wird dieser Fall anschließend abgeprüft, und der Punkt wird, sollte er ein Element der strikt aktiven Menge sein, relaxiert. Nach der Relaxation ist sein skaliertes Residuum 0. Sein Residuum liegt also nicht mehr über τ und er ist somit nicht länger Element der strikt aktiven Menge. Durch die Relaxation haben sich jedoch die Residuen seiner Nachbarn geändert. Obwohl nicht sicher ist, daß auch nur einer dieser Nachbarpunkte nun über der Toleranz τ liegt, werden – der Einfachheit halber– alle in die aktive Menge $\tilde{S}$ aufgenommen. Somit beinhaltet $\tilde{S}$ weiterhin mindestens alle Punkte deren skaliertes Residuum über τ liegen und ist somit eine aktive Menge.

3.2 V. Daums Implementierung des Algorithmus

Eine Implementierung dieses Algorithmus entstand im Rahmen der Studienarbeit „Runtime-controlled adaptivity in multigrid methods“ von V. Daum [Dau01]. Während dieser Arbeit wurde unter anderem die Implementierung der aktiven Menge genauer untersucht.

Für die punktweise adaptive Relaxation ergaben sich folgende Ergebnisse.

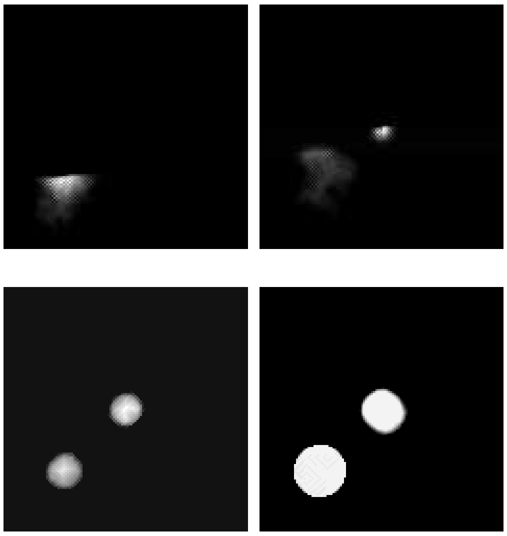


Abbildung 3.1: Die unterschiedliche Verlauf bei Implementierung eines Stacks (oben) beziehungsweise eines Fifo (unten) (2-Peaks-Problem; entnommen aus [Dau01]).

1. Der in [Rü93] vorgestellte Algorithmus kann selbstständig Bereiche identifizieren, an denen vermehrte Glättungen notwendig sind. Er „findet“ also „points of high interest“ (siehe auch Abbildung 3.1).

2. Die *Performanz* des Algorithmus hängt maßgeblich von der Implementierung der aktiven Menge $\mathcal{S}$ ab.

   - Die aktive Menge durch eine *sortierte Liste* darzustellen, ist zu aufwendig. Hier ist offensichtlich, daß der Vorteil des adaptiven Verfahrens gegenüber einem Standard-GS – nämlich die verringerte Anzahl von Relaxation per Punkt – durch den erhöhten Aufwand, den die Mengenverwaltung benötigt, zunichte gemacht wird.
   - Die Darstellung der aktiven Menge mittels eines *Stacks* ist ebenfalls uneffektiv, da sich der Algorithmus hier an einer Stelle „festbeißt“ und erst nach erfolgreichem Glätten/Lösen dieser Stelle zu einem anderen Bereich im Gitter weiterspringt, in dem Arbeit zu leisten ist.
   - Eine einfache und dennoch effiziente Möglichkeit, die aktive Menge zu implementieren, ist ein *Fifo*.
     ("[...] it is very probable that a point which has just been added to the set, has been relaxed more often than a point which has been in the set for a long time. Therefore the fifo is in most cases a small step in the direction of the ideal algorithm that was mentioned in the beginning [...] which always relaxes the point with the biggest residual first, and obviously this seems to pay off." [Dau01])

3. Leider läuft der Algorithmus nur bei bestimmten *Modellproblemen* wirklich effizient.

   - Für Modellprobleme mit *gleichmäßig verteilten Residuen* bringt der adaptive Algorithmus keinen Vorteil. Für ein glattes Laplace-Problem benötigt er sogar mehr Relaxationen pro Punkt als ein Standard-Gauß-Seidel.

- Benutzt man Modellprobleme, für die der den Algorithmus geeigneter ist, kann die Anzahl der Relaxationen pro Punkt signifikant reduziert werden (vergleiche dazu Tabelle 3.1).

4. Trotzdem bleibt der implementierte Algorithmus *zeitlich* gesehen immer noch hinter dem Standardverfahren. Dies liegt zum einen an der Tatsache, daß bei Gauß-Seidel durch die festgelegte Zugriffsreihenfolge ein gutes Prefetching möglich ist und zudem eine hohe räumliche Lokalität gegeben ist (siehe dazu auch [HP03]). Zum anderen verlängert der zusätzliche Aufwand, der durch die Verwaltung der aktiven Menge entsteht, die Laufzeit des Verfahren merklich.

⇒ "In practice one would not do all this on a point per point basis but rather divide the grid into patches and then do all of the above per patch and not per point. If these patches were tailored to the size of the machine caches then the overhead from the set operations probably will not make a too big effect anymore compared to the time needed to load another patch into the caches. " [Dau01, aus den Schlußbemerkungen]

Aktive Menge	Globales Residuum	Anzahl Punkt Relaxationen	
		Punkt-adaptiver GS	Standard-GS
Stack	$6.2E-03$	548532	661289
Fifo	$1.4E-02$	90931	306451

Tabelle 3.1: Eine Auswahl der Ergebnisse für unterschiedliche Implementierungen der aktiven Menge im punktweise adaptiven Fall (2-Peaks-Problem; entnommen aus [Dau01]).

Zusammenfassung:
Da dieser Algorithmus durchaus dazu in der Lage ist, Singularitäten oder andere Punkte, die höheren Arbeitsaufwand erfordern, zu finden, sollte die *Erweiterung auf Patches* die Nachteile des Algorithmus, namentlich den erhöhten Arbeitsaufwand, den ineffizienten Cache-Zugriff und die damit verbundene längere Laufzeit, ausgleichen, beziehungsweise zu einem robusteren und trotzdem nicht zu langsamen Glätter für die Verwendung im Multigrid-Kontext führen.

3.3 H. Lötzbeyers Erweiterung auf Patches

Die Idee, das Gitter in Patches zu unterteilen, wurde bereits von H. Lötzbeyer in seiner Diplomarbeit „Parallele adaptive Mehrgitterverfahren“ verfolgt [Lö96].

Im Zuge dieser Diplomarbeit wurden einige der „Adaptivitäten“ aus [Rü93] implementiert und getestet. Leider lag der Hauptaugenmerk auf der Implementierung des adaptiv verfeinerten Gitters und weniger auf dem Patch-adaptiven Glätter. So liegen keine Ergebnisse für einen adaptiven Glätter, wie in Abschnitt 3.1 beschrieben, vor.

Der von H. Lötzbeyer implementierte und untersuchte Glätter arbeitet auf Patches, kommt aber gänzlich ohne aktive Menge aus. So geht dieser Glätter lediglich p-mal durch die Liste aller Patches und relaxiert jedes Patch dabei q-mal mithilfe eines Standard-Gauß-Seidel-Verfahrens mit lexikographischer Ordnung. Der Pseudocode für diesen Algorithmus, der im folgenden als *patchweise arbeitender Gauß-Seidel* bezeichnet wird, ist unter Algorithmus 7.1 beschrieben.

Zwar ist dieser Glätter nicht mit dem Patch-adaptiven GS vergleichbar, der im Rahmen der Studienarbeit implementiert wurde, einige Ergebnisse lassen sich aber trotzdem transferieren. Eines dieser Ergebnisse ist die in [Lö96] gezeigte Zeitersparnis, die durch das mehrfache Glätten eines Patches aufgrund der dadurch gegebenen Cache-Zugriffsoptimierung erzielt werden kann. (Eine detaillierte Übersicht diverser Ansätze zur Cachoptimierung findet sich in [KW03].) Da unser Code nicht auf Performanz untersucht wurde, weil keinerlei Versuche der Optimierung (wie zum Beispiel loop unrolling, loop reordering, etc.) unternommen werden konnten, sind die Zeitmessungen aus [Lö96] von besonderer Bedeutung. Auch die schlechtere Konvergenz durch die Aufteilung des Gebiets in einzelne Patches und die damit verbundene Ansammlung großer Komponenten des Residuums an den Patchgrenzen, hebelt die Geschwindigkeitsvorteile des Algorithmus nicht zwingend aus.

Die Tests, die auf das Verhältnis von Glättungsgeschwindigkeit und Genauigkeit abzielen, wurden von uns in Abschnitt 7.1 mit ähnlichen Erfolgen wiederholt.

Außerdem gelang es uns, die starre Patchstruktur, die von H. Lötzbeyer als zwingend erforderlich angesehen wurde, aufzubrechen. Zwar sind unsere Tests ebenfalls mit einer regulären Aufteilung in Patches ohne Überlappungen gerechnet worden, wie aber in Abschnitt 6.2.2 gezeigt wird, sind auch weit flexiblere Einteilungen möglich.

Aufgrund der bisher vorliegenden Ergebnisse wurden von uns zwei verschiedene Verfahren entwickelt, die den punktweise adaptiven GS (Algorithmus 3.1) auf Patches arbeiten lassen. Beide Varianten des Lösers, die vor allem als Vor- und Nachglätter in einem Multigridkontext Anwendung finden sollen, sind im nun folgenden Kapitel ausführlich beschrieben.

Kapitel 4

Patch-adaptiver Glätter

4.1 Der Basisalgorithmus

Um den punktweise adaptiven GS (Algorithmus 3.1) auf Patches zu übertragen, sind einige Vorüberlegungen und Designentscheidungen notwendig. Eine eins-zu-eins Übertragung des punktweisen Algorithmus auf Patches findet sich in Algorithmus 4.1 „patchweise adaptive Relaxation Version 1“. Ein Patch ist ein Rechteck, welches einen Teil des Gitters u überdeckt. Es kennt seine linke untere $((\mathcal{P}_{i_{min}}, \mathcal{P}_{j_{min}}))$, und rechte obere Ecke $((\mathcal{P}_{i_{max}}, \mathcal{P}_{j_{max}}))$ (siehe Abbildung 4.1).

Algorithmus 4.1 patchweise adaptive Relaxation Version 1

Require: tolerance τ, initial guess $u^{(0)}$, activeSet $\tilde{S}(\tau, \mathcal{P})$
Require: set of patches $\mathcal{P} = \mathcal{P}_{u^{(0)}} = \{\mathcal{P}_0, \dots \mathcal{P}_m\}$

while $\tilde{S}(\tau, \mathcal{P}) \neq \emptyset$ **do**
  pick patch $\mathcal{P}_k \in \tilde{S}(\tau, \mathcal{P})$
  delete $\mathcal{P}_k$ from $\tilde{S}(\tau, \mathcal{P})$
  $\bar{r}'(k) \leftarrow$ `computeScaledPatchResidual` $(\mathcal{P}_k)$
  if $\bar{r}'(k) > \tau$ **then**
    `relaxPatch` $(\mathcal{P}_k)$
    `putNeighbourPatchesInActiveSet` $(\tilde{S}(\tau, \mathcal{P}), \mathcal{P}_k)$
  end if
end while

4.2 Gemeinsamkeiten von Punkt- und Patch-adaptiver Relaxation

Auch im patchweisen Fall verwendet man als initiale aktive Menge am einfachsten die Menge aller Patches, da diese eine natürliche Obermenge zur strikt aktiven Menge ist. Obwohl dies durch den Algorithmus nicht fest vorgegeben ist, hat es sich in Praxistests als tauglich erwiesen.

Aktive Menge $\tilde{S}(\tau, \mathcal{P})$ (Definition)

Die aktive Menge $\tilde{S}(\tau, \mathcal{P})$ zu einer Toleranz τ und einer Menge von Patches $\mathcal{P}$ ist definiert als

$$\tilde{S}(\tau, \mathcal{P}) := \{\mathcal{P}_k | \; \exists u_{(i,j)} \in \mathcal{P}_k \text{ mit } |\bar{r}_{(i,j)}| \geq \tau\}.$$

Sowohl die Auswahl eines Patches $\mathcal{P}_k$ aus der aktiven Menge $\tilde{S}(\tau, \mathcal{P})$ als auch das Löschen dieses Patches funktioniert genauso wie im Punkt-adaptiven Fall. Ein Unterschied zwischen den beiden Versionen besteht darin, daß durch die Aufteilung des Gitters der Größe $n \times n$ in m Patches ($m \ll n^2$), die Anzahl der in der aktiven Menge zu verwaltenden Elemente stark reduziert ist. Der Overhead, der für Auswahl, Löschen und Hinzufügen von Elementen in die aktive Menge im Punkt-adaptiven Fall durchaus ins Gewicht fällt, ist somit im Patch-adaptiven Fall vernachlässigbar.

Die beiden Funktionen `computeScaledPatchResidual()` und `relaxPatch()` sind äquivalent zum punktweisen Verfahren: Beide Funktionen gehen über alle Elemente $u_{(i,j)}$ des Patches und berechnen deren skaliertes Residuum beziehungsweise relaxieren jeden Punkt. Bei der Berechnung des skalierten Residuums des Patches verwendet man am einfachsten die Maximumsnorm. Eine Beschreibung der einzelnen Teiloperationen findet sich in den Algorithmen 4.2 und 4.3.

Algorithmus 4.2 `computeScaledPatchResidual` $(\mathcal{P}_k)$

Require: $i_{min}, i_{max}, j_{min}, j_{max}$ for Patch $\mathcal{P}_k$

$\bar{r}' = 0.0$
for (`i` = i_{min} ; `i` $\leq i_{max}$; `i++`) **do**
  for (`j` = j_{min} ; `j` $\leq j_{max}$; `j++`) **do**
    $\bar{r}_{(i,j)} \leftarrow$ `computeScaledResidual` $(u_{(i,j)})$
    if $\bar{r}_{(i,j)} \geq \bar{r}'$ **then**
      $\bar{r}' \leftarrow \bar{r}_{(i,j)}$
    end if
  end for
end for
`return` $\bar{r}'$

Algorithmus 4.3 `relaxPatch` $(\mathcal{P}_k)$

Require: $i_{min}, i_{max}, j_{min}, j_{max}$ for Patch $\mathcal{P}_k$

for (`i` = i_{min} ; `i` $\leq i_{max}$; `i++`) **do**
  for (`j` = j_{min} ; `j` $\leq j_{max}$; `j++`) **do**
    `relax` $(u_{(i,j)})$
  end for
end for

Auch die Funktion `putNeighbourPatchesInActiveSet()` kann so implementiert werden, daß einfach alle in die neun Richtungen des Sterns angrenzenden Patches der aktiven Menge hinzugefügt werden. Wird innerhalb des Algorithmus das Patch nur ein einziges Mal geglättet, ist nicht sichergestellt, daß sein (Patch-)Residuum unter die Grenze τ gefallen ist. Aus diesem Grund muß in diesem Fall auch das Patch $\mathcal{P}_k$ in die aktive Menge eingefügt werden. Es fällt erst aus der Menge, wenn bei einem erneuten Aufruf sein dann zu berechnendes Residuum unter der Toleranz liegt. Dieses Verfahren kann allerdings – wie im folgenden noch ausführlich gezeigt (siehe Abschnitt 4.3.2) – auch weitaus effizienter programmiert werden, um weitere Rechenoperationen einzusparen.

Da nach Terminierung des Algorithmus die aktive Menge leer ist, sind alle skalierten Patchresiduen unter der Toleranz τ. Da die Maximumsnorm verwendet wurde und das gesamte Gitter von Patches überdeckt ist, ist somit auch die Maximumsnorm des Residuum des Lösungsvektors u unter der Toleranz τ.

4.3 Unterschiede zwischen Punkt- und Patch-adaptiver Relaxation

Obwohl sich der Algorithmus relativ leicht auf Patches übertragen läßt, gibt es einige Änderungen, die sich im Patch-adaptiven Fall anbieten:

4.3.1 Mehrmaliges Glätten eines Patches

Dies ist einer der Gründe, warum das Verfahren überhaupt auf Patches übertragen wurde. Man geht davon aus, daß ein Patch, befindet es sich erst einmal im Cache, relativ effizient mehrmals geglättet werden kann. Da der Zugriff auf Daten im Cache viel schneller erfolgen kann als auf Daten, die erst von der Festplatte in den Arbeitsspeicher und weiter in die Register geholt werden müssen, benötigt ein zweites Glätten eines Patches nur ungefähr 40 % der Zeit des ersten Glättungsvorgangs. (Weitere Informationen über Cachehierarchien und Möglichkeiten ihrer Ausnutzung finden sich in [Han98] und [KW03].) Dies konnte in Versuchen bereits gezeigt werden (vergleiche [Lö96]). Für die Anzahl der Glättungen, die auf einem Patch ausgeführt werden, gibt es mindestens zwei verschiedene Möglichkeiten:

1. Die erste Variante glättet jedes Patch, auf dem gearbeitet wird, genau x mal.
2. Die zweite glättet jedes Patch solange, bis es unter die Toleranz τ gefallen ist.

Beide Verfahren haben ihre Vor- und Nachteile:

Im ersten Fall muß das Patch selbst nach dem Glätten wieder in die aktive Menge aufgenommen werden, da nicht sicher ist, ob sein Residuum unter der geforderten Toleranz liegt. Dieses Verfahren hat allerdings den Vorteil, daß sich Probleme, die für Gebietszerlegungsverfahren bekannt sind, nicht so stark wie im zweiten Fall auswirken. Glättet man ein Patch so kennt dieses Patch die eigentlichen Dirichletränder des Gebiets nicht. Während es geglättet wird betrachtet es seine angrenzenden Ränder als Dirichletränder. Da die Unbekannten der Patchränder sich jedoch – gerade am Anfang des Iterationsverfahrens – stark von ihrer exakten Lösung u^* unterscheiden verbessert ein exzessives Glätten des Patches unter eine Toleranz τ trotz des hohen Aufwands den Fehler der momentanen Lösung $u^{(k)}$ kaum.

Der zweite Fall, in dem ein Patch solange geglättet wird, bis sein skaliertes Residuum – respektive die Maximumsnorm desselben – unter die gewünschte Toleranz τ fällt, hat den Vorteil, daß das Patch während der Funktion `putNeighbourPatchesInActiveSet()` nicht selbst wieder in die aktive Menge zurückgeschrieben werden muß. Da man in diesem Fall auch eine genaue Aussage über die Maximumsnorm des Patchresiduums treffen kann, läßt sich das Aktivierungskriterium weiter verfeinern (siehe Abschnitt 4.3.2). Zwar haben wir beide

Verfahren implementiert, für die Tests aus Effizienzgründen allerdings lediglich das zweite Verfahren verwendet. Der Pseudocode dazu sieht wie in Algorithmus 4.4 aus.

Algorithmus 4.4 `relaxPatch` $(\mathcal{P}_k,\ \tau)$

Require: $i_{min}, i_{max}, j_{min}, j_{max}$ for Patch $\mathcal{P}_k$
while $\bar{r}'_{\mathcal{P}_k} > \tau$ **do**
  for (i = i_{min} ; i $\leq i_{max}$; i++) **do**
    for (j = j_{min} ; j $\leq j_{max}$; j++) **do**
      `relax` $(u_{(i,j)})$
    end for
  end for
  $\bar{r}'_{\mathcal{P}_k} \leftarrow$ `computeScaledPatchResidual` $(\mathcal{P}_k)$
end while

4.3.2 Eingeschränkte Aktivierung der Nachbarn

Wir gehen im folgenden davon aus, daß die Funktion `relaxPatch ()` wie in Algorithmus 4.4 beschrieben, implementiert ist. Zu dem Zeitpunkt, zu dem `putNeighbourPatchesInActiveSet()` aufgerufen wird, ist die Maximumsnorm des Residuums von $\mathcal{P}_k$ somit $\leq \tau$.

1. Die naheliegendste Möglichkeit, die aus der Punktweise-adaptiven Relaxation kommt, ist, einfach alle Nachbarpatches in die aktive Menge zu nehmen. Dies ist in Algorithmus 4.5 festgehalten.

Algorithmus 4.5 `putNeighbourPatchesInActiveSet` $(\mathcal{P}_k)$ `Version 1`

Require: Neighbours $\mathcal{P}_{\text{Conn}} = \{\mathcal{P}_{NW}, \mathcal{P}_N, \mathcal{P}_{NE}, \mathcal{P}_E, \mathcal{P}_{SE}, \mathcal{P}_S, \mathcal{P}_{SW}, \mathcal{P}_W\}$ for Patch $\mathcal{P}_k$
for all $\mathcal{P}_x \in \mathcal{P}_{\text{Conn}}$ **do**
  $\tilde{S}(\tau, \mathcal{P}) \leftarrow \tilde{S}(\tau, \mathcal{P}) \cup \{\mathcal{P}_x\}$
end for

2. Eine Verfeinerung des Auswahlkriteriums wäre, nur die Patches in die aktive Menge aufzunehmen, deren Residuum kleiner als die geforderte Toleranz ist. Diese zweite Möglichkeit benötigt allerdings genau die gleiche Anzahl Rechenoperationen wie Algorithmus 4.5. Die Berechnung des Residuums findet in diesem Fall ebenfalls statt, nämlich dann, wenn das Patch aus der aktiven Menge ausgewählt wurde (vergleiche Algorithmus 4.1 Zeile 4).

3. Das hier zuletzt vorgestellte Verfahren ist sehr komplex, hat dafür aber auch das Potential, eine ganze Reihe von Berechnungen einzusparen. Für dieses Aktivierungskriterium benötigt man zwei zusätzliche Toleranzen τ_{patch} und τ_δ. Für diese gilt folgende Gleichung

$$\tau_{\text{patch}} + \tau_\delta = \tau. \tag{4.1}$$

   Während der Relaxation eines Patches wird dieses so lange relaxiert, bis sein skaliertes Residuum unter τ_{patch} gefallen ist. Um den Verlauf dieses Verfahren leichter verstehen zu können, siehe Abbildungen 4.1, 4.2 und 4.3.

Nach dem Glätten des Patches werden in `putNeighbourPatchesInActiveSet` $(\mathcal{P}_k)$ zuerst alle Patches auf eine notwendige Aktivierung überprüft, die nur einen einzigen Berührungspunkt zu dem gerade relaxierten Patch $\mathcal{P}_k$ besitzen. Dies sind seine Nachbarn $\mathcal{P}_{NW}$, $\mathcal{P}_{NE}$, $\mathcal{P}_{SW}$ und $\mathcal{P}_{SE}$. Für diese muß im Berührungspunkt das skalierte Residuum explizit berechnet werden. Dies ist notwendig, da diese Residuen von verschiedenen Patches abhängig sind. Ist eines der angrenzenden Patches bereits relaxiert worden, besteht die Möglichkeit, daß die Maximumsnorm des skalierten Residuums am Berührungspunkt unter τ blieb. Wird nun ein zweites angrenzendes Patch geglättet, reicht es nicht aus nur die Änderung in seinem skalierten Residuum zu berechnen, da man etwaige Änderungen aus vorangegangenen Patchrelaxationen nicht kennt. Deshalb muß an diesen Stellen das skalierte Residuum explizit berechnet werden. Ist es größer als τ, wird das zugehörige Patch in die aktive Menge $\tilde{S}(\tau, \mathcal{P})$ aufgenommen.

Anschließend werden die Nachbarn $\mathcal{P}_N$. $\mathcal{P}_W$, $\mathcal{P}_S$ und $\mathcal{P}_E$ betrachtet. Für diese muß an den äußeren Randpunkte ganz im Osten und Westen (für die Nord- und Südgrenze) respektive im Norden und Süden (für die westlich und östlich angrenzende Punktreihe) ebenfalls ein skaliertes Residuum explizit berechnet werden. Ist das skalierte Residuum größer als τ muß das entsprechende Patch in die aktive Menge aufgenommen werden. (*Beispiel:* Die Punkte $u_{(\mathcal{P}^N_{i_{min}}, \mathcal{P}^N_{j_{min}})}$ und $u_{(\mathcal{P}^N_{i_{min}}, \mathcal{P}^N_{j_{max}})}$ für die Nordgrenze.)

Für die restlichen (dazwischenliegenden) Punkte der Grenze, reicht es aus, die Änderung des Residuums δ zu betrachten. Diese Änderung läßt sich aus der Differenz der momentanen Werte der Punkte zu den Werten dieser Punkte *vor* der letzten Aktivierung des Patches durch diesen Rand berechnen. Da sich die inneren Punkte des Patches nicht geändert haben, ergibt sich die Änderung durch die Änderungen δ_1 δ_2 und δ_3 der drei Punkte, auf die mithilfe des Sterns zugegriffen wird. Die alten Werte (vor der letzten Aktivierung) sind für jede der vier Richtungen in einem Vektor `backup` gespeichert. Ist die Änderung δ größer als τ_δ, muß das Patch in die aktive Menge aufgenommen werden. (*Beispiel:* Die Punkte $u_{(\mathcal{P}^N_{i_{min}}, \mathcal{P}^N_{j_{min}+1})}$ bis $u_{(\mathcal{P}^N_{i_{min}}, \mathcal{P}^N_{j_{max}-1})}$ für die Nordgrenze.)

Da im Inneren eines Patches die Maximumsnorm des skalierten Residuums kleiner als τ_{patch} ist und da an den Grenzen der Betrag des Residuums höchstens $\tau_{\text{patch}} + \tau_\delta$ ist, gilt für das skalierte Residuum des gesamten Gitters

$$|\bar{r}| \leq \tau.$$

Wie `putNeighbourPatchesInActiveSet`$(\mathcal{P}_k)$ im Pseudocode aussieht, beschreibt Algorithmus 4.6.

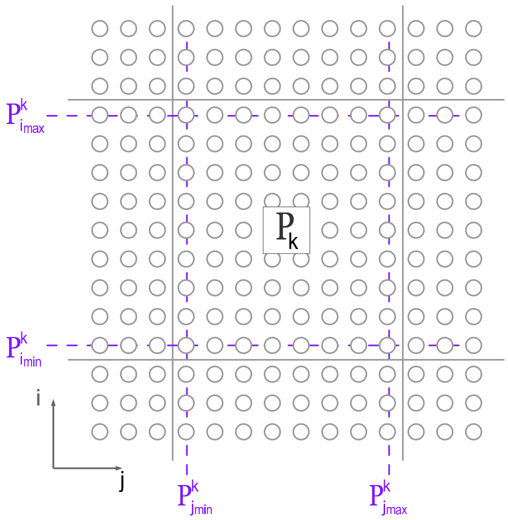


Abbildung 4.1: Die Lage der Werte i_{min}, i_{max}, j_{min} und j_{max} für Patch $\mathcal{P}_k$.

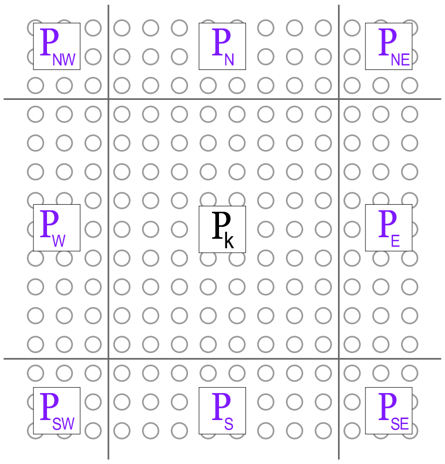


Abbildung 4.2: Die Nachbarschaft $\mathcal{P}_{\text{Conn}}$ eines Patches $\mathcal{P}_k$.

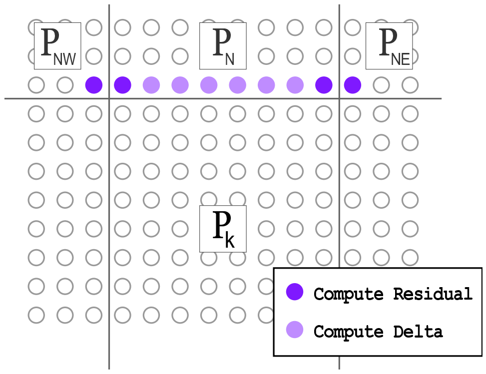


Abbildung 4.3: Die Punkte an der nördlichen Grenze, für die das Residuum explizit berechnet werden muß bzw. diejenigen, für die eine Berechnung ihrer Änderung δ ausreichend ist.

Algorithmus 4.6 `putNeighbourPatchesInActiveSet` $(\mathcal{P}_k)$

Require: Neighbours $\mathcal{P}_{\text{Conn}} = \{\mathcal{P}_{NW}, \mathcal{P}_N, \mathcal{P}_{NE}, \mathcal{P}_E, \mathcal{P}_{SE}, \mathcal{P}_S, \mathcal{P}_{SW}, \mathcal{P}_W\}$ for Patch $\mathcal{P}_k$

`// Angrenzende Patches mit Berührungspunkt`
for $\mathcal{P}_{NW}$ **do**
  $i = \mathcal{P}^{NW}_{i_{min}}, j = \mathcal{P}^{NW}_{j_{max}}$
  if $\bar{r}_{(i,j)} > \tau$ **then**
    $\tilde{S}(\tau, \mathcal{P}) \leftarrow \tilde{S}(\tau, \mathcal{P}) \cup \{\mathcal{P}_{NW}\}$
  end if
end for
for $\mathcal{P}_{NE}, \mathcal{P}_{SW}, \mathcal{P}_{SE}$ **do**
  $\ldots$
end for
`// Angrenzende Patches mit Berührungsrand`
for $\mathcal{P}_N$ **do**
  $i = \mathcal{P}^N_{i_{min}}$
  `// Eckpunkt links`
  if $\bar{r}_{(i,\mathcal{P}^N_{j_{min}})} > \tau$ **then**
    $\tilde{S}(\tau, \mathcal{P}) \leftarrow \tilde{S}(\tau, \mathcal{P}) \cup \{\mathcal{P}_N\}$
  end if
  `// Eckpunkt rechts`
  if $\bar{r}_{(i,\mathcal{P}^N_{j_{max}}} > \tau$ **then**
    $\tilde{S}(\tau, \mathcal{P}) \leftarrow \tilde{S}(\tau, \mathcal{P}) \cup \{\mathcal{P}_N\}$
  end if
  `// Punkte dazwischen`
  $\delta = 0.0$
  for $(j = \mathcal{P}^N_{j_{min}} + 1; j < \mathcal{P}^N_{j_{max}}; j{+}{+})$ **do**
    $\delta_1 = \texttt{backup}_{u_{(i-1,j-1)}} - u_{(i-1,j-1)}$
    $\delta_2 = \texttt{backup}_{u_{(i-1,j)}} - u_{(i-1,j)}$
    $\delta_3 = \texttt{backup}_{u_{(i-1,j+1)}} - u_{(i-1,j+1)}$
    $\delta = (\delta_1 \cdot a^{(i,j)}_{SW} + \delta_2 \cdot a^{(i,j)}_{S} + \delta_3 \cdot a^{(i,j)}_{SE})/a^{(i,j)}_{C}$
    if $\delta > \tau_\delta$ **then**
      $\tilde{S}(\tau, \mathcal{P}) \leftarrow \tilde{S}(\tau, \mathcal{P}) \cup \{\mathcal{P}_N\}$
      `break`
    end if
  end for
end for
for $\mathcal{P}_E, \mathcal{P}_W, \mathcal{P}_S$ **do**
  $\ldots$
end for

Kapitel 5

Verwendete Modellprobleme

Durch den modularen objektorientierten Aufbau des Codes ist es relativ einfach, zusätzliche Modellprobleme zu implementieren. Die Datei `mg_init.C` enthält alle Informationen über die Matrix A, die rechte Seite f und die Startnäherung $u^{(0)}$ des Lösungsvektors u. Sie ist somit die einzige Datei, die im allgemeinen geändert werden muß.

5.1 Glattes Modellproblem

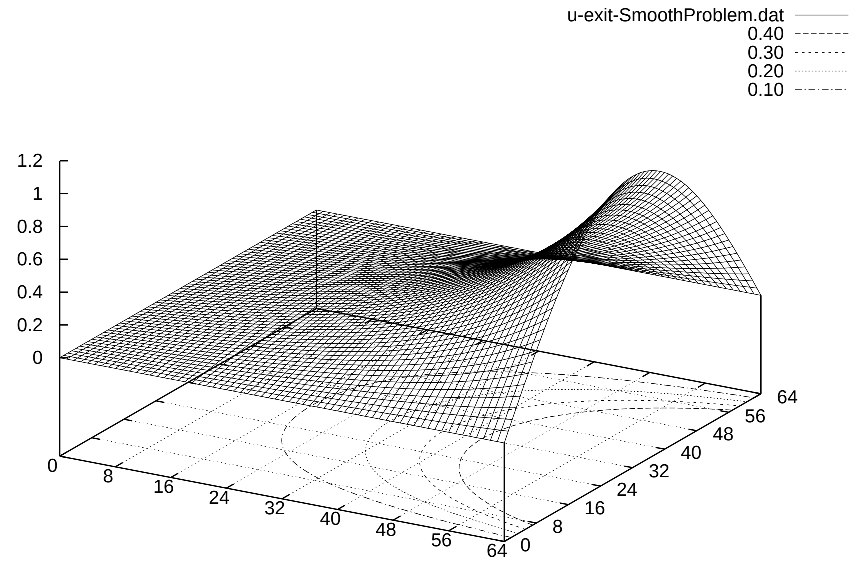


Abbildung 5.1: Die Lösung des glatten Problems (siehe Gleichung 5.1).

$$
\begin{aligned}
-\Delta u &= 0.0 && \text{in } \Omega = (0,1)^2 \\
u &= 0.0 && \text{auf } \delta\Omega \text{ für } x = 0; y = 0 \vee y = 1 \\
u &= \sin(\pi y) \cdot \sinh(\pi) && \text{auf } \delta\Omega \text{ für } x = 1
\end{aligned} \tag{5.1}
$$

Initialisierung:
$u^{(0)} = 0.0$ innerhalb von Ω und auf dem Rand $\delta\Omega$ für $x = 0; y = 0; y = 1$.
$u^{(0)} = \sin(\pi y) \cdot \sinh(\pi)$ für $x = 1$.

Analytische Lösung
$u^* = \sin(\pi y) \cdot \sinh(\pi x)$ in Ω .

Obwohl von vornherein klar war, daß der Patch-adaptive Glätter bei glatten Modellproblemen keinen Performance-gewinn erzielen kann – eventuell sogar im Vergleich zu den Standard-GS-Glättern langsamer sein würde – implementierten wir dieses Modellproblem. Es diente somit lediglich als Vergleich zu dem in diesem Kapitel ebenfalls beschriebenen L-Gebiet-Problem.

5.2 Glattes Modellproblem – Konvergenzproblem

$$\begin{aligned} -\Delta u = &\quad 0.0 \quad \text{in } \Omega = (0,1)^2 \\ u = &\quad 0.0 \quad \text{auf } \delta\Omega \end{aligned} \tag{5.2}$$

Initialisierung:
$u^{(0)} = \sin(\pi x) \cdot \sin(\pi y)$ innerhalb von Ω .
$u^{(0)} = 0.0$ auf $\delta\Omega$.

Analytische Lösung
$u^* = 0.0$ in Ω .

Relativ schnell stellte sich heraus, daß als Vergleichsgröße der einzelnen Glätter ihre Konvergenzrate betrachtet werden sollte. Erste Versuche, in denen auf dem eigentlichen glatten Modellproblem (Gleichung 5.1) eine Vielzahl von V-Zyklen gerechnet wurden, um die asymptotische Konvergenzrate zu bestimmen, scheiterten daran, daß schon nach ungefähr 15 Zyklen die Lösung so genau berechnet war, daß Rundungsfehler die Bestimmung der Konvergenzrate verhinderten. Die asymptotische Konvergenzrate, die man auch als „worst case Konvergenzrate" bezeichnen kann, war bei diesen Versuchen in der Reduktion der Residuen nicht zu erkennen. Sie sollte, bestimmt durch den Spektralradius der Iterationsmatrix, für glatte Modellprobleme ungefähr bei 0.1 liegen.

Mithilfe eines einfachen Tricks war es uns aber doch möglich, Konvergenzraten für das glatte Problem zu ermitteln. Da die asymptotische Konvergenzrate durch den Spektralradius der Iterationsmatrix festgelegt ist, ändert sie sich nicht mit einer Änderung der Randbedingungen oder der Startlösung $u^{(0)}$. Somit können beide beliebig variiert werden.

Um die Lösung anschaulicher werden zu lassen, entschieden wir uns dazu, die exakte Null auszurechnen, also $-\Delta u = 0$ zu betrachten. Der algebraische Fehler ist in diesem Fall die Abweichung vom Nullvektor. So sieht man in einem Plot der Näherungslösung u direkt den aktuellen Fehler und hat – im Vergleich zu der Norm des Residuums – somit ein genaueres und vor allem visuelles Feedback über die Güte der Lösung.

Um Rundungsfehlern zu entgehen, die sich ab einer Größenordnung von 10^{-14} bemerkbar machen, wurde außerdem vor jedem V-Zyklus die momentane Näherungslösung hochskaliert, so daß die Norm des Residuums auf dem feinsten Gitter immer 1.0 betrug. Der Pseudocode für dieses „artificial scaling" findet sich in Algorithmus 5.1.

Nach genügend V-Zyklen mit „artificial scaling“ konvergiert u nun nicht mehr gegen die Lösung des Modellproblems sondern gegen den Eigenvektor, der zum betragsgrößten Eigenwert der Iterationsmatrix gehört. Würde man diesen Eigenvektor schon vorher kennen und als Startlösung für $u^{(0)}$ wählen, könnte man bereits im ersten V-Zyklus die *worst case Konvergenzrate* des Multigridverfahrens für den Operator A und die gewählten Verfahren zur Glättung und Grobgitterkorrektur sehen. Dies resultiert aus der Tatsache, daß das Mehrgitterverfahren für genau diese Startlösung die meisten Schwierigkeiten hat.

Algorithmus 5.1 „`artificial Scaling`“

$\bar{r} \leftarrow$ `computeResidualAtFinestLevel()`
for all $u_{(i,j)}$ on finest level **do**
    $u_{(i,j)} \leftarrow \frac{1}{r} \cdot u_{(i,j)}$
end for
`do V-Cycle`

Eine Übersicht der ermittelten Konvergenzraten findet sich in Abschnitt 7.2.2 in Tabelle 7.6.

5.3 L-Gebiet-Problem

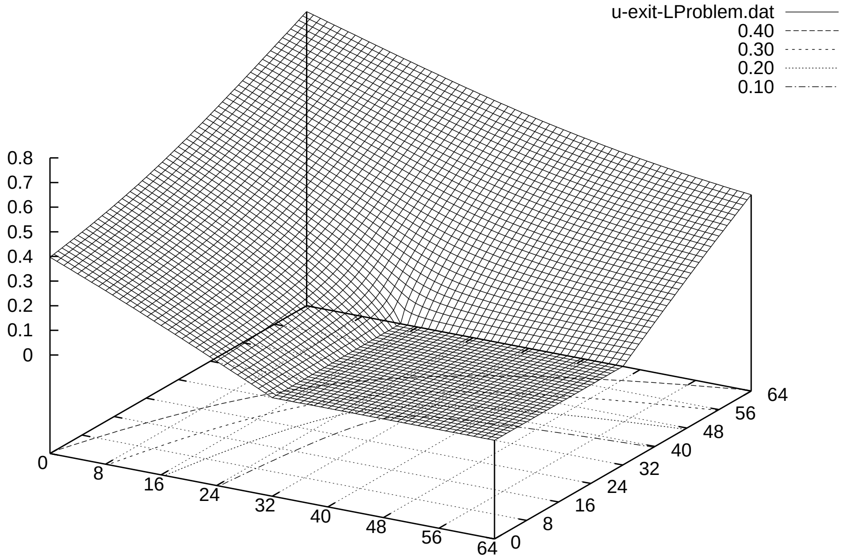


Abbildung 5.2: Die Lösung des L-Gebiet Problems (siehe Gleichung 5.3).

$$
\begin{aligned}
-\Delta u &= \quad 0.0 \qquad \text{in } \Omega = (-0.5, 0.5)^2 \setminus (0, 0.5) \times (-0.5, 0) \\
u &= \sin(\tfrac{2}{3}\varphi) \cdot r^{\frac{2}{3}} \quad \text{auf } \delta\Omega
\end{aligned}
\tag{5.3}
$$

Initialisierung:
$u^{(0)} = 0.0$ innerhalb von Ω.
$u^{(0)} = \sin(\frac{2}{3}\varphi) \cdot r^{\frac{2}{3}}$ auf $\delta\Omega$.

Analytische Lösung
$u^{*} = \sin(\frac{2}{3}\varphi) \cdot r^{\frac{2}{3}}$ auf $\delta\Omega$

Das L-Gebiet-Problem ist eines der Standardprobleme, welche zur Betrachtung von Singularitäten im Operator herangezogen werden. Dadurch daß das Problem lediglich in einem L-förmigen Gebiet definiert ist und an seinem Mittelpunkt ($r = 0$) künstlich per Dirichletrand auf 0 gehalten wird, ergibt sich in seinem Graph eine Singularität, die durch die unendliche Steigung aus diesem Punkt heraus anschaulich wird.

Da dieses Problem eben nicht auf einem rechteckigen Gebiet definiert ist, müssen leider auch Änderungen außerhalb von `mg_init.C` vorgenommen werden. Diese beinhalten zum Beispiel die Einschränkung von `gs_lex()`, `gs_redblack()` und `computeResidual()` (alle aus `level.C`) auf Punkte, für die $i > \frac{\texttt{size}}{2}$ oder $j < \frac{\texttt{size}}{2}$ gilt (wobei `size`$=2^l$ mit $l \in \mathbf{N}$).

Für den Patch-adaptiven Glätter ist es wichtig, daß die Aufteilung der Patches so vorgenommen wird, daß alle inneren Punkte von Patches überdeckt werden, so daß kein Patch auch Punkte aus dem undefinierten Teil des Gebiets enthält. Dies erreicht man durch das in Algorithmus 5.2 vorgestellte Verfahren. Für die Dirichletrandpunkte an der „inneren“ Grenze entschieden wir uns dazu, diese Punkte nicht explizit als Randpunkte zu kennzeichnen, sondern sie stattdessen in angrenzende Patches zu verschieben, die den Teil des Gebiets überdecken, der nicht definiert ist. Dies hat den Vorteil, daß die entsprechenden inneren Punkte ohne Änderungen auf ihren Wert zugreifen können, sie selbst aber nie verändert werden und somit genau das Verhalten von Dirichletrandpunkten an den Tag legen.

Algorithmus 5.2 `PatchLevel::setPatchBoundaries()`

```
int patchSizeX = (size+1)/numPatchesX; // size= 2^l
int patchSizeY = (size+1)/numPatchesY;
for int k= 0; k < numPatchesY; k++ do
  iMin[k]= k * patchSizeY;
end for
jMin[_numPatchesY/2]+=1;
for int k= 0; k < numPatchesX; k++ do
  iMin[k]= k * patchSizeX;
end for
```

Mit der oben beschriebenen Einteilung der Patches können die Änderungen am Patch-adaptiven Glätter einfach in der Implementierung der aktiven Menge vorgenommen werden und bestehen lediglich darin, die Patches, die in dem nicht definierten Bereich liegen, nie zu aktivieren. Damit die Werte in diesem Bereich auch für die Interpolation und Restriktion korrekt sind, wird bei der ersten Initialisierung der Gitter in `mg_init.C` darauf geachtet, daß auf allen Ebenen in dem Viertel, das nicht zum Definitionsbereich gehört, die Werte für $u_{(i,j)}$ auf 0.0 gesetzt werden. Somit muß die Funktion `gs_adaptive()` aus `patchLevel.C` nicht verändert werden. Lediglich die Routine `put()` in `aSet.C` wird, wie unter Algorithmus 5.3 beschrieben, geändert.

Wie erste Testläufe mit dem L-Gebiet-Problem zeigten, scheint die Konvergenzrate des Problems – wie erwartet – über den 0.1 für Standardmehrgitterverfahren zu liegen. Durch die

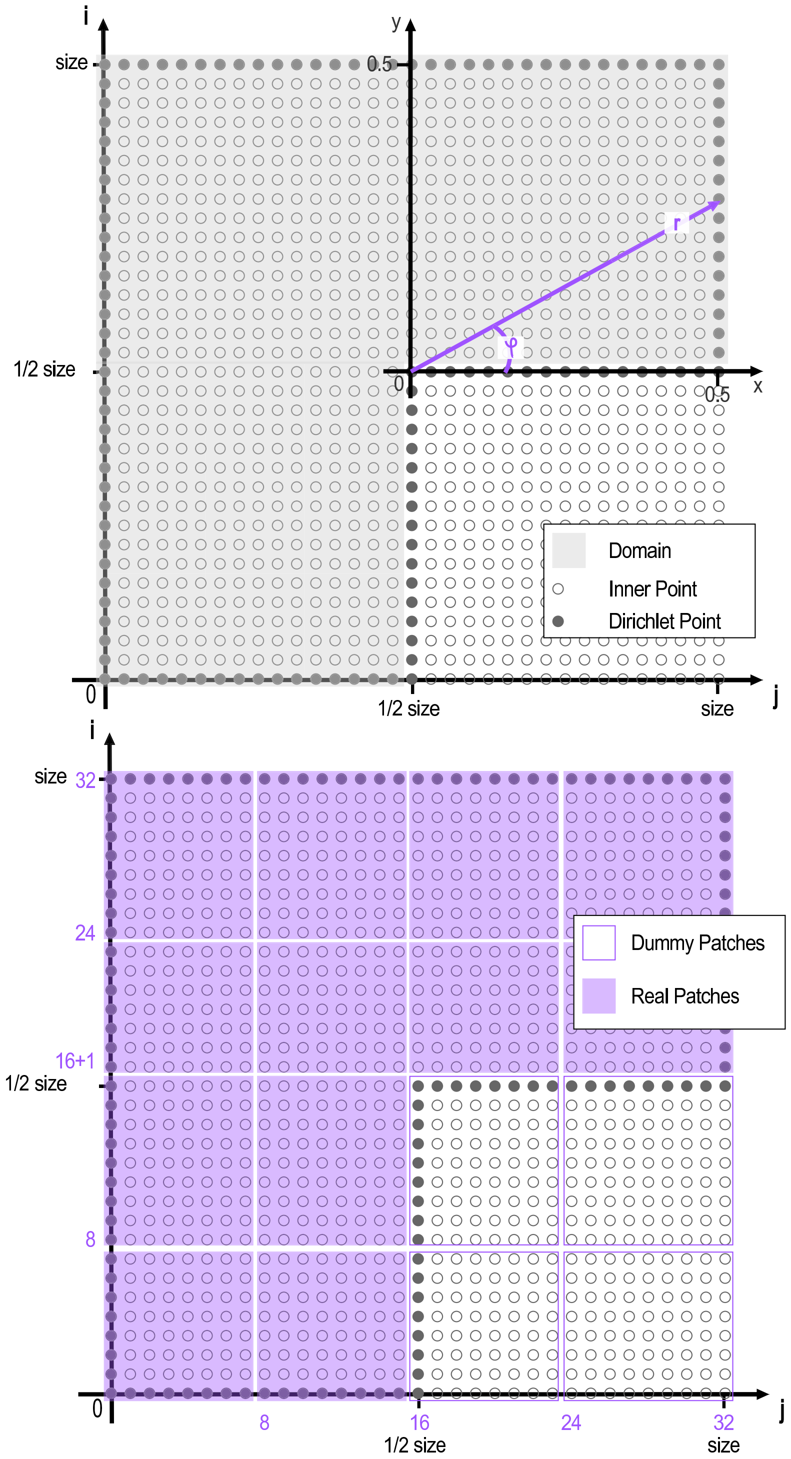


Abbildung 5.3: Der Definitionsbereich des L-Gebiet-Problems und die Aufteilung der Patches.

Algorithmus 5.3 `ASet::put( int element )`

if `element` is out of Ω **then**
  do not put element in list
else
  put element in list
end if

Einschränkung des Gebiets auf die drei Viertel eines Einheitsquadrats fehlen dem Operator A eine Reihe von Zeilen. Diese sind letzten Endes die Verantwortlichen für die schlechtere Konvergenzrate. Auch für das L-Gebiet-Problem ändert sich an der asymptotischen Konvergenzrate nichts, wenn man es – analog zu der im vorherigen Abschnitt beschriebenen Vorgehensweise – ändert, um die Konvergenzrate des Verfahrens bestimmen zu können.

5.4 L-Gebiet-Problem – Konvergenzproblem

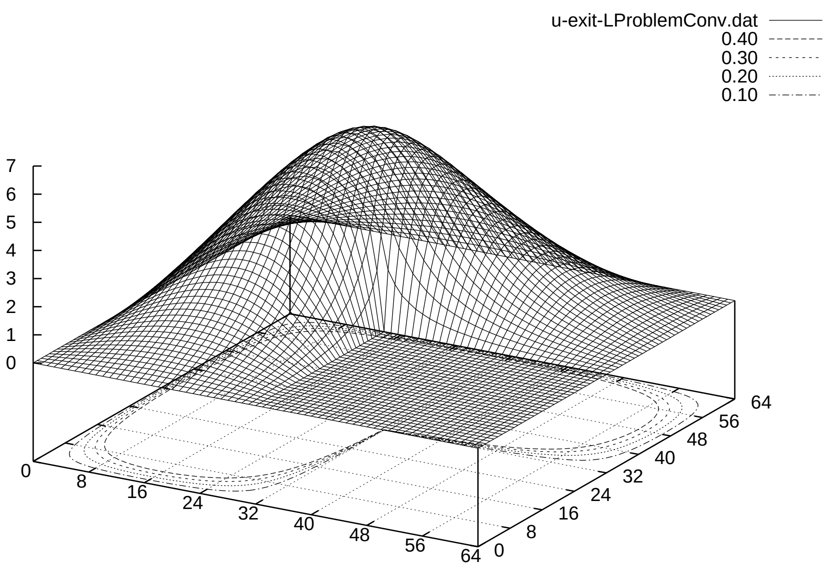


Abbildung 5.4: Der Eigenvektor zum betragsgrößten Eigenwert der Iterationsmatrix für das L-Gebiet-Konvergenzproblem.

$$\begin{aligned} -\Delta u &= 0.0 \quad \text{in } \Omega = (-0.5, 0.5)^2 \setminus (0, 0.5) \times (-0.5, 0) \\ u &= 0.0 \quad \text{auf } \delta\Omega \end{aligned} \tag{5.4}$$

Initialisierung:
$u^{(0)} = 1.0$ innerhalb von Ω.
$u^{(0)} = 0.0$ auf $\delta\Omega$.

Analytische Lösung
$u^* = 0.0$ in Ω .

Da auch für das L-Gebiet-Problem Konvergenzraten gemessen werden sollten, mußte dieses Problem ebenfalls in ein äquivalentes Problem umgeschrieben werden. Auch in diesem Fall berechnet man $-\Delta u = 0$, was als exakte Lösung die Null hat. Somit kann man auch hier an dem Plot des Lösungsvektors den Fehler ablesen. Als Startnäherung muß man lediglich darauf achten, nicht schon die exakte Lösung zu verwenden. Wir haben uns dafür entschieden, die inneren Punkte mit 1.0 zu initialisieren.

Des weiteren kam auch hier „artificial scaling“ zum Einsatz, um die asymptotische Konvergenzrate möglichst exakt ermitteln zu können. Das „artificial scaling“ muß im Vergleich zum glatten Problem (siehe Algorithmus 5.1) nicht verändert werden, da die Punkte, die außerhalb des Gebiets liegen, allesamt auf 0.0 bleiben. Abbildung 5.4 zeigt den durch „artificial scaling“ und eine hohe Anzahl von V-Zyklen ermittelten Eigenvektor zum betragsgrößten Eigenwert der Iterationsmatrix für das L-Gebiet-Konvergenzproblem und einen rot-schwarzen-Glätter.

Ein Vergleich der ermittelten Konvergenzraten zwischen glattem und L-Gebiet-Problem zeigt – wie erwartet – deutliche Einbrüche der Konvergenzrate für das L-Gebiet-Problem (vergleiche Tabelle 7.6 aus Abschnitt 7.2.2).

Kapitel 6

Die Implementierung

6.1 class ASet

Die nach den Ergebnissen aus [Dau01] am einfachsten zu beantwortende Frage, ist die nach der Implementierung der aktiven Menge. Teilt man das Gitter in ausreichend große Patches auf, ist die Anzahl der in der Menge zu verwaltenden Elemente im Vergleich zur Punkt-adaptiven Relaxation äußerst gering. Da sich während des Laufes die Residuen der einzelnen Patches häufig ändern, kann eine sortierte Liste jedoch nur mit dem erhöhten Aufwand einer regelmäßigen Residuumsberechnung aller Patches realisiert werden. Dieser Aufwand ist ungerechtfertigt, da bereits Ergebnisse vorliegen, wonach eine Fifo ebenfalls zu einer Art sortierter Liste wird, da die „ältesten“ Elemente meistens die mit den schlechtesten Residuen sind (vergleiche dazu die Ergebnisse aus [Dau01] vorgestellt in Abschnitt 3.2).

Durch die Kapselung der Implementierung der aktiven Menge in der Klasse `ASet`, ist eine einfache Reimplementierung einer anderen Verwaltungsart (zum Beispiel ein Stack) unabhängig vom Rest des Programmes und somit leicht möglich.

Die Klasse *ASet* stellt im wesentlichen folgende Funktionen zur Verfügung:

```
put()
get()
initialFill()
```

Dabei ist `initialFill()` die Funktion, die den „initial guess“ der aktiven Menge $\tilde{S}$ in `activeSet` füllt. Dieser ist, wie in Kapitel 4 beschrieben, im einfachsten Fall die Menge aller Patches, da diese nicht zu groß ist und eine natürliche Wahl darstellt.

6.2 class Patch

Wie sieht nun ein Patch-Objekt aus? In unserer Implementierung sind Patches lediglich „Fenster“ über dem Gitter. Dies bedeutet, daß die eigentlichen Werte der einzelnen Punkte $u_{(i,j)}$ in einem großen Vektor gespeichert sind, auf den alle Patches zugreifen können. So können auf diesem Vektor auch ohne Performanzverluste Standardglätter, wie zum Beispiel ein Gauß-Seidel-Verfahren, verwendet werden. Eine genaue Beschreibung der einzelnen Elemente der Klasse `Patch` findet sich als Doxygen-Output auf der beiliegenden CD im Unterverzeichnis `/doc`. Grob betrachtet speichert ein Patchobjekt folgende Werte:

- seine linke untere und rechte obere Ecke: `int i_min, i_max, j_min, j_max;`
- die ID des Levels im Multigrid, zu dem es gehört: `int levelId;`
- die ID des Patches auf diesem Level: `int id;`
- die IDs seiner Nachbarn: `vector<int> neighbourPatches;`
- die Toleranzen τ und τ_{delta}: `double scalResUB, scalResDelta;`

Dabei ist zu beachten, daß die Punkte $u_{(i_{min},j_{min})}$, $u_{(i_{max},j_{min})}$, $u_{(i_{min},j_{max})}$ und $u_{(i_{max},j_{max})}$ die vier Ecken des rechteckigen Patches darstellen und somit noch zum Patch gehören (siehe dazu auch Abbildung 4.1). Da das Patch Zugriff auf den gesammten Vektor u hat, kann es während der Funktion `putNeighbourPatchesInActiveSet()` auch auf die Werte der angrenzenden Patches zugreifen, um angrenzende Punktresiduen einfach selbst zu berechnen.

6.2.1 Wie werden Patches angeordnet?

Die einzige Forderung für die Einteilung eines Gitters in Patches ist die, daß das Gitter komplett mit Patches überdeckt sein muß. Das heißt, jeder Punkt $u_{(i,j)}$ muß mindestens einem Patch $\mathcal{P}_k$ zugeordnet sein.

Im Programmcode haben wir eine Variable eingefügt, die die Anzahl der Patches und deren ungefähre Größe festlegt. Diese Variable heißt `SINGLE_PATCH_LEVEL_ID` und ist in der Datei `pas_globals.h` zu finden. Sie ist abhängig von der Anzahl der Level, die für das Mehrgitterverfahren angelegt werden, und welche durch die Variable `NUM_LEVELS` in `mg_globals.h` festgelegt ist. Die Variable `SINGLE_PATCH_LEVEL_ID` gibt dabei an, ab welchem Level Patches angelegt werden und somit auch, ab welchem Level potentiell der adaptive Glätter `gs_adaptive()` angewandt wird. Die Größe des Levels, für den `SINGLE_PATCH_LEVEL_ID` = `LevelID` gilt, ist die ungefähre Größe der einzelnen Patches, die ab dem nächstfeineren Level angelegt werden. Da die Patchgröße primär von der Cachegröße abhängen sollte, sind die Patches auf allen Levels gleich groß, es variiert lediglich ihre Anzahl.

Beispiel: Legt man insgesamt 10 Levels an, erhalten diese die `IDs` 0 bis 9, wobei Level 0 der gröbste und Level 9 der feinste Level ist. Dabei hat Level 9 die in der Variablen `DIM` festgelegte Anzahl von Intervallen pro Dimension. Ist nun zum Beispiel `SINGLE_PATCH_LEVEL_ID` = 3, werden auf den Levels 0 bis 3 keine Patches angelegt. Auf diesen Levels wird mittels rot-schwarzem Gauß-Seidel geglättet. Die Größe von Level 3 ist die ungefähre Größe für alle Patches, die auf den Leveln 4 bis 9 angelegt werden. So besteht Level 4 aus 4 Patches, Level 5 bereits aus 16 Patches, und so weiter.

6.2.2 Alternative Einteilungen

Im Moment kümmert sich die Funktion `PatchLevel::setPatchBoundaries(numPatchesX, numPatchesY)` darum, daß möglichst Patches gleicher Größe generiert werden. Während der Entstehungsphase haben wir allerdings auch mit sich überlappenden Patches und mit unterschiedlichen Patchgrößen auf ein und demselben Level experimentiert. Grundsätzlich scheint eine einfache Erweiterung des Programmes durch andere Patchaufteilungen möglich und bietet eventuell weitere Effizienzvorteile. Hierzu muß lediglich sichergestellt sein, daß die im Vektor `neighbourPatches` gespeicherten IDs zu den am Patch angrenzenden Punkten gehören.

6.3 class Hierarchy

Die Klasse `Hierarchy` ist die Kernkomponente des Mehrgitterverfahrens, da sie Zugriff auf die einzelnen Levels hat und damit Leveloperationen wie den Glätter oder die Residuumsberechnung anstoßen kann. Sie besteht lediglich aus `NUM_LEVELS` Pointern auf Level-Objekte. Abhängig davon, ob der Patch-adaptive Glätter im Programm verwendet wird, zeigen diese Pointer auf Objekte vom Typ `PatchLevel` oder, falls keine Patches notwendig sind, auf Objekte vom Typ `Level`.

6.4 class Level

Die Klasse `Level` ist das Herzstück der Standardglätter `gs_redblack()` und `gs_lex()`. Diese Klasse hat Zugriff auf das Gitter u ebenso wie auf die rechte Seite f und den Operator A (beide zusammen gespeichert in einem Objekt der Klasse `StencilRHSArray`). Da die Klasse `PatchLevel` von `class Level` abgeleitet ist, kann auch ein PatchLevel-Objekt die Standardglätter und die Berechnung des Residuums auf dem gesamten Gitter (`computeResidual()`) aufrufen.

6.5 class PatchLevel

Diese Klasse ist, wie schon erwähnt, von `class Level` abgeleitet. Als Verfeinerung zur Level-Klasse besitzt sie außer den Zeigern auf u, A und f zusätzlich einen Vektor aller zum Level gehörenden Patches $\mathcal{P}$, eine aktive Menge $\tilde{S}$ und die Toleranzen τ (`= scalResUB`) sowie τ_δ (`= scalResDelta`). Diese Klasse beinhaltet den hier vorgestellten Patch-adaptiven Glätter `gs_adaptive()`.

6.6 mg.C

Diese Datei beinhaltet das eigentliche Mehrgitterverfahren. Hier kann unter verschiedensten Möglichkeiten (FMG, V-Cycles, Correction Scheme, FAS, Galerkin Produkt, ...) ausgewählt werden. Welches Verfahren verwendet wird, stellt man per Makro in der Datei `mg_globals.h` ein. Für den Patch-adaptiven Glätter können – beziehungsweise müssen – noch weitere Variablen eingestellt werden, diese finden sich in der Datei `pas_globals.h`.

6.7 mg_direct_solve.C

Diese Datei implementiert den direkten Löser, der im allgemeinen auf dem gröbsten Gitter angewendet wird. Hierzu wird eine LU-Zerlegung gerechnet, die lediglich die geeignete LAPACK-Routine aufruft. (Zur Verwendung der LAPACK-Routinen siehe [ABB+99]. Eine detaillierte Beschreibung der LU-Zerlegung findet sich unter [HS02].)
Bemerkung: Das L-Gebiet-Problem läßt sich im Rahmen unserer Implementierung nicht mittels direktem Löser rechnen, hier ist es notwendig, die Anzahl Levels so zu wählen, daß der gröbste Level aus nur einer (Pseudo-)Unbekannten besteht. In diesem Modellproblem ist die Unbekannte in Wirklichkeit ein Dirichletrand mit dem Wert 0.0 und somit direkt die Lösung.

6.8 `mg_init.C`

Die Datei `mg_init.C` kapselt die Implementierung des Modellproblems, so daß auch andere Modellprobleme mithilfe einer einfachen Reimplementierung dieser Datei schnell zum Einsatz kommen können. Das L-Gebiet-Problem benötigt leider auch Änderungen außerhalb dieser Datei, weshalb sich ein abgeleiteter Code auf der CD im Unterverzeichnis `/prog/L_model_problem` befindet. (Eine Auflistung der vorgenommenen Änderungen findet sich in Abschnitt 5.3.)

6.9 `scalRes.h` – oder: Die Wahl der Toleranzen

Die Wahl der Toleranzen für den Patch-adaptiven Glätter war eine der am schwierigsten zu beantwortenden Fragen während dieser Studienarbeit. Schon in [Dau01] konnte diese Frage, trotz einiger Experimente dazu, nicht zufriedenstellend beantwortet werden. Im Mehrgitterkontext scheint sie noch schwieriger beantwortbar zu sein, da sich die Toleranzen auf allen Levels unterscheiden und nach jedem V-Zyklus angepaßt werden müssen. Versuche, die Toleranz aufgrund der Residuen eines Standardglätters zu bestimmen (vergleiche Abschnitt 7.2.3), waren nicht erfolgreich. So wurde letzten Endes die Datei `scalRes.h` angelegt, in der für jeden Level eine Schranke τ definiert ist. Diese Werte werden zur Compilezeit eingelesen und als initiale Toleranzen bei der Generierung der `PatchLevel`-Objekte verwendet.

Die Datei muß dazu vorher mit geeigneten Werten gefüllt werden. Eine Möglichkeit ist, die Werte für einen Standardglätter nach dem ersten V-Zyklus zu ermitteln und diese zu verwenden. Hier wird dann in dem V-Zyklus mit Patch-adaptivem Glätter für jeden Level gefordert, daß sein Residuum unter dem des V-Zyklus mit Standardglätter liegt. Eine zweite Möglichkeit besteht darin, die Datei mit den momentanen Residuen zu versehen und noch vor dem ersten Aufruf des Glätters mithilfe von `PatchLevel::resetScalRes(double factor)` die Residuen auf jedem Level auf $\text{factor} \cdot \text{res}_{\text{current}}$ zu drücken. So kann man bequem eine Konvergenzrate einstellen.

Die Wahl der Toleranz τ_δ erfolgt in Abhängigkeit von τ. Wir haben nur mit folgenden Fällen experimentiert: $\tau_\delta = 0.5 \cdot \tau$ oder $\tau_\delta = 0.2 \cdot \tau$. Auch hier könnten weitere Experimente ein noch besseres Verhältnis von Konvergenzrate und aufgewendeter Arbeit erzielen.

Kapitel 7

Testreihen für Single- und Multilevel

7.1 Test: Heikos Versuche

In einer ersten Testreihe wurde versucht, Ergebnisse aus [Lö96] zu rekonstruieren.

7.1.1 Versuchsaufbau

Für diese Testreihe war ein weiterer Glätter notwendig, der zwar auf Patches arbeitet, die Möglichkeiten eines adaptiven Glätters allerdings nicht voll ausnutzt. Dieser Glätter, als `PatchLevel::gs_patch(int sweeps, int relaxationsPerPatch)` implementiert, geht `sweeps`-mal über alle Patches und veranlaßt diese, sich jeweils `relaxationsPerPatch`-mal zu glätten (Pseudocode: siehe Algorithmus 7.1).

Algorithmus 7.1 Patchweise arbeitenden Gauß-Seidel
`gs_patch(int sweeps, int relaxationsPerPatch)`

for $i = 0$ to `sweeps` **do**
  for all Patches **do**
    `relaxPatchXTimes(relaxationsPerPatch)`
  end for
end for

Bei den nachfolgenden Tests handelt es sich im wesentlichen um Versuche, die in der Diplomarbeit von H. Lötzbeyer in Abschnitt 4.1 („Glättungsalgorithmen“) dokumentiert sind. Da durch die von ihm verwendete adaptive Gitterstruktur ein Test mit genau denselben Ausgangsparametern nicht möglich war, mußten einige Änderungen am Versuchsaufbau vorgenommen werden:

Modellproblem Zum Testzeitpunkt stand leider die Implementation des L-Gebiet-Problems noch nicht zur Verfügung, sodaß wir das Standard-Laplace-Problem verwendeten. Abbildung 5.1 zeigt die Lösung des Modellproblems (Abschnitt 5.1).

Gitterhierarchie Da aus der Dokumentation nicht genau ersichtlich ist, wie groß das verwendete Gitter war, versuchten wir, dieses möglichst gut anzunähern. Im Text ist von 1055 Patches und einer Verfeinerung auf 5 Gitterebenen die Rede. Wir verwenden nun ein feinstes Gitter von 1023x1023 Unbekannten, legen 7 Gitterebenen an, so daß das

Level id	dim	Größe Patches	Anzahl Patches auf Level	Anzahl Patches gesamt
6	1024	16x16	4096	5461
5	512	16x16	1024	1365
4	256	16x16	256	341
3	128	16x16	64	85
2	64	16x16	16	21
1	32	16x16	4	5
0	16	16x16	1	1

Tabelle 7.1: Verwendete Gitterstruktur.

gröbste Gitter genau 15×15 Unbekannte besitzt, also aus lediglich einem Patch besteht (Eine genaue Auflistung der Gitterhierarchie findet sich in Tabelle 7.1).

Patchstruktur Gleich geblieben ist die (ungefähre) Patchgröße von 16x16 Punkten.

Glätter Als Glätter kamen – abhängig vom Versuch – `gs_patch`, `gs_lex` und `gs_redblack` zum Einsatz. Auf dem gröbsten Level wurde mittels LU-Zerlegung gelöst.

Meßgrößen Desweiteren verwenden wir als Vergleichsgröße die L_2-Norm des skalierten Residuums nach dem ersten durchlaufenen V-Zyklus. Zeitmessungen konnten am bisherigen Code nicht sinnvoll durchgeführt werden, da er nicht performanceoptimiert ist.

Konvergenzrate Die gemessenen Konvergenzraten für die einzelnen Glätter stammen aus einem separaten Versuch. Hier wurde der Code so abgeändert, daß vor jedem V-Zyklus die Näherung des Lösungsvektors $u^{(k)}$ durch die L_2-Norm des Residuums geteilt wurde. So wird die Norm des Residuum immer wieder auf 1.0 hochskaliert und verfälschende Rundungsfehler ausgeschlossen. `NUM_ITER` wurde zusätzlich auf 200 gesetzt, um eine stabile „worst-case“-Konvergenzrate zu erhalten. (Eine genaue Beschreibung dieses Konvergenz-Modellproblems findet sich in Abschnitt 5.2.)

Für alle Testreihen stehen auch die gewonnenen (ausführlichen) Daten zur Verfügung. Diese finden sich auf der beigelegten CD unter `/Prog/Results_Heiko/`.

```
mg_globals:
 #define NUM_LEVELS 7
 #define DIM 1024
 #define NUM_ITER 30
pas_globals:
 #define SINGLE_PATCH_LEVEL_ID 0
script.sh:
 for SMOOTHER in PATCH_GS LEX_GS RB_GS
 for PRESMOOTH (=POSTSMOOTH) in 1 2 3 4 5 6 7
```

Abbildung 7.1: Die wichtigsten Makros aus den Definitionsskripten für diesen Test.

7.1.2 Die erste Testreihe

Die erste Testreihe untersucht das Verhältnis der beiden Variablen `sweeps` und `relaxationsPerPatch` zueinander. Erstere bestimmt, wie oft jeder Patch zum Glätten aufgerufen wird, letztere wie oft bei einem Aufruf von `relaxPatchXTimes()` das Patch nacheinander geglättet wird. Offensichtlich ist für einen größeren Wert von `relaxationsPerPatch` eine gewisse Zeitersparnis zu erwarten, solange die Daten für einen Patch in den Cache passen. Allerdings wird das Problem der Gebietszerlegung bei größerem `relaxationsPerPatch` stärker. So muß ein sinnvolles Verhältnis von Glättungen per Gitter zu Glättungen per Patch gefunden werden. Leider ist der vorliegende Code im Rahmen dieser Studienarbeit nicht auf Performanz optimiert worden, weswegen Zeitmessungen nicht aussagekräftig sind und lediglich auf die guten Ergebnisse der Zeitmessungen in [Lö96] verwiesen werden kann.

Die Ergebnisse dieser Testreihe sind in Tabelle 7.2 aufgelistet. Dabei gibt die dritte Spalte die Norm des Residuums nach dem ersten V-Zyklus an, während dessen sowohl zum Vor- als auch zum Nachglätten `sweeps`×`relaxationsPerPatch`-Anzahl Relaxationen pro Punkt durchgeführt wurden. Obwohl als Vergleichsgröße nur die L_2-Norm des Residuums nach dem ersten V-Zyklus zur Verfügung steht, kann man erkennen, daß für 2 `sweeps` à 3 `relaxationsPerPatch` ein gutes Gleichgewicht zwischen Cacheoptimierung und Konvergenzrate vorliegt. Nach diesen 2x3 Glättungen fällt die Norm des Residuums auf $2,8 \cdot 10^{-2}$.

Zum Vergleich: Glättet man das Gitter mit 6 Vor- bzw. Nachglättungsschritten einer Standard-GS-Implementierung, ergibt sich nach dem ersten V-Zyklus eine L_2-Norm des Residuums von $2,6 \cdot 10^{-2}$. Auch nach dem achten V-Zyklus befinden sich die Normen des Residuums für beide Glätter in derselben Größenordnung (siehe Datei `VglRatioSweepsRelaxPerPatchL2Norm.ps` auf der CD).

`sweeps` = Glättungen per Gitter	`relaxationsPerPatch` = Glättungen per Patch	L_2-Norm des Residuums nach dem 1. V-Zyklus
1	6	0.0969995
2	3	0.0283023
3	2	0.0267771
6	1	0.0261294

Tabelle 7.2: Variation des Verhältnisses von `sweeps` und `relaxationsPerPatch` (vgl. Tabelle 4.1 Seite 38 in [Lö96]).

7.1.3 Die zweite Testreihe

Die zweite Testreihe aus [Lö96] wurde, da sie lediglich auf Geschwindigkeitsbetrachtungen ausgelegt ist, von uns nicht wiederholt.

7.1.4 Die dritte Testreihe

Die dritte Testreihe war für uns besonders interessant, da für sie außer Zeitmessungen auch die Messung der L_2-Norm des Fehlers in [Lö96] dokumentiert war. So war ein direkter Vergleich zwischen unseren und den Ergebnissen in [Lö96] möglich. In diesem Experiment geht es darum, die Anzahl der Vor- und Nachglättungsschritte zu variieren und dann die Güte der Lösung nach dem ersten V-Zyklus zwischen verschiedenen Glättern zu vergleichen. Als Gesamtgitterglätter verwenden wir die beiden implementierten, zum einen `gs_redblack()` und zum anderen `gs_lex()`. Als Patchglätter kommt der oben beschriebene `gs_patch()` zum Einsatz. Für letzteren wird lediglich `relaxationsPerPatch` variiert, `sweeps` wird auf 1 festgehalten. Dabei gilt: `relaxationsPerPatch`= $\sharp$(Vor- bzw. Nachglättungen). Leider konnten wir nicht die Norm des Fehlers sondern nur die Norm des Residuums messen, da im allgemeinen die exakten Lösungen – und damit der Fehler – nicht bekannt sind.

Daß sich durch die Verwendung des patchweise arbeitenden Gauß-Seidels für eine hohe Anzahl von Glättungen pro Patch die Probleme der Gebietszerlegung stark bemerkbar machen, indem sich die hohen Werte des Residuums vor allem an den Patchgrenzen sammeln, sieht man in den folgenden beiden Tabellen (Tabelle 7.3 und Tabelle 7.4): Mißt man lediglich die Maximumsnorm des Residuums, ist der Unterschied zwischen dem patchweise arbeitendem GS und dem lexikographischen bzw. rot-schwarzen enorm. Verwendet man hingegen die diskrete L_2-Norm, werden die Abstände zwischen den Glättern kleiner und der Geschwindigkeitszuwachs des `gs_patch()` damit von größerem Vorteil.

Vor- bzw. Nachglättungen	`gs_redblack()` MaxNormOfResidual	`gs_patch()` MaxNormOfResidual	`gs_lex()` MaxNormOfResidual
1	0.049522200	0.0289126	0.0289126
2	0.005303510	0.0199242	0.00514988
3	0.002201190	0.0159889	0.00207894
4	0.001256750	0.0143712	0.0013664
5	0.000808794	0.0136392	0.000904904
6	0.000584076	0.0132692	0.000648605
7	0.000436276	0.0130643	0.000486102

Tabelle 7.3: Vergleich der einzelnen Glätter (Maximumsnorm)
(vgl. Tabelle 4.3 Seite 39 in [Lö96]).

Vor- bzw. Nachglättungen	`gs_redblack()` L_2-NormOfResidual	`gs_patch()` L_2-NormOfResidual	`gs_lex()` L_2-NormOfResidual
1	0.815277	0.810538	0.810538
2	0.114724	0.232321	0.160725
3	0.0533958	0.15146	0.0830925
4	0.0329619	0.120818	0.0527254
5	0.0226458	0.105465	0.035894
6	0.0166312	0.0969995	0.0261294
7	0.0127949	0.0919476	0.0199733

Tabelle 7.4: Vergleich der einzelnen Glätter (Diskrete L_2-Norm)
(vgl. Tabelle 4.3 Seite 39 in [Lö96]).

Vor- und Nachglättungen	`gs_redblack()` Konvergenzrate	`gs_patch()` Konvergenzrate	`gs_lex()` Konvergenzrate
1	0.11	0.19	0.19
2	0.060	0.102	0.100
3	0.041	0.068	0.066
4	0.031	0.051	0.050
5	0.025	0.042	0.040
6	0.021	0.037	0.033
7	0.018	0.034	0.028

Tabelle 7.5: Konvergenzraten des MG-Verfahrens unter Verwendung der verschiedenen Glätter im glatten Modellproblem.

7.2 Test: Konvergenz des PAS

7.2.1 Versuchsaufbau

Modellproblem Abhängig vom Test kamen die beiden Konvergenzraten-Probleme (siehe Abschnitt 5.2 und 5.4) einzeln oder gemeinsam zum Einsatz.

Gitterhierarchie Für die Tests verwendeten wir ein Gitter mit 1023x1023 Unbekannten. Im glatten Modellproblem bestand die Gitterhierachie aus insgesamt 7 Gittern, das gröbste Gitter somit aus 15x15 Unbekannten.
Das L-Gebiet-Problem verlangt ein gröbstes Gitter mit einem einzigen inneren Punkt, der für dieses Problem keine Unbekannte sondern ein Dirichletrandpunkt mit Wert 0.0 ist. So ergibt sich für dieses Problem eine Gitterhierarchie von insgesamt 10 Levels.

Glätter Als Glätter kamen vorwiegend `gs_lex()` und `gs_redblack()` zum Einsatz. Lediglich für den abschließenden Performanztest (Abschnitt 7.2.4) verwendeten wir auch den Patch-adaptiven Glätter `gs_adaptive()` in seiner Version mit subtilem Aktivierungskriterium (vergleiche Abschnitt 4.3.2, insbesondere Algorithmus 4.6).

Patchstruktur Der Patch-adaptive Glätter verwendet das Makro `SINGLE_PATCH_LEVEL_ID = 4`. Dies bedeutet für seinen Einsatz im L-Gebiet, daß Level 4 die Größe der Patches bestimmt (Level 0: gröbstes Gitter, ein innerer Punkt; Level 9: feinstes Gitter 1023x1023 innere Punkte; $\rightarrow$ Level 4: 31x31 innere Punkte). Dieser Level besteht aus genau einem Patch, weswegen er der feinste Level ist, auf dem noch der Standardglätter zum Einsatz kommt. Alle feineren Level bestehen aus ungefähr 32x32 Punkte großen und damit aus

$$(2^{\,\texttt{LEVEL\_ID} - \texttt{SINGLE\_PATCH\_LEVEL\_ID}})^2$$

vielen Patches und werden durch `gs_adaptive()` geglättet.
(Zur Verwendung der Konstante `SINGLE_PATCH_LEVEL_ID` siehe Abschnitt 6.2.1.)

Verwendete Norm Für die folgenden Versuche wurde die Maximumsnorm $\|.\|_\infty$ verwendet. Für die diskrete L_2-Norm gelten äquivalente Ergebnisse, welche hier lediglich aus Platzgründen nicht aufgeführt sind.

Meßgrößen Als Vergleichsgröße der einzelnen Glätter dient vor allem deren ermittelte Konvergenzrate (zur Ermittlung der Konvergenzrate siehe Abschnitt 5.2 für das glatte Modellproblem und Abschnitt 5.4 für das L-Gebiet Problem).
Diese ist für einen Standardglätter durch die Kombination von Vor- und Nachglättungsschritten festgelegt. Die Anzahl seiner Punktrelaxation berechnet sich somit als

$$(\sharp\text{Vor-} + \sharp\text{Nachglättungen}) * (\sharp\text{Unbekannte}_{\texttt{LEVEL 0}} + \ldots + \sharp\text{Unbekannte}_{\texttt{LEVEL (NUM\_LEVELS -1)}})$$

Für den Patch-adaptiven Glätter ist es schwieriger, eine Konvergenzrate oder die Anzahl der Punktrelaxationen zu ermitteln. Mithilfe der Toleranzen τ_{patch} , τ_δ und τ kann man quasi jede Konvergenzrate für diesen Glätter erzwingen. Die Anzahl der Punktrelaxationen läßt sich hingegen nur schwer festlegen, da diese stark von den gewählten Toleranzen abhängt. Hier gilt es demzufolge, ein vernünftiges Verhältnis von erreichter Konvergenzrate und gemessener Punktrelaxationen zu erreichen.

Auch für diese Testreihen stehen eine Reihe von Rohdaten zur Verfügung die sich auf der CD in den Verzeichnissen `/Prog/ConvergenceTest/Results` und `/Prog/L_Problem/ConvergenceTest/Results/` finden.

7.2.2 Ermittlung der Konvergenzen im L-Gebiet

```
script.sh:
 for SMOOTHER in RB_GS LEX_GS
 for PRESMOOTH in  0 1 2 3 4
 for POSTSMOOTH in 0 1 2 3 4
```

Abbildung 7.2: Das Skript zur Ermittlung der Konvergenzraten.

```
mg_globals.h:
 #define DIM 1024
 #define NUM_LEVELS 10
 #define NUM_ITER 100
 #define USE_MAX_NORM
```

Abbildung 7.3: Die Einstellungen für die Tests im Überblick.

Eine Auflistung der Ergebnisse findet sich in Tabelle 7.6.

Die Bestimmung der Konvergenzraten für das L-Gebiet zeigt: Die Konvergenzrate des Mehrgitterverfahrens mit Standardglättern bricht von unter 0.1 (für ein glattes Problem) auf bis zu 0.54 im L-Gebiet zusammen. Dieser Effekt war durch die „Singularität“ im Operator erwartet worden. Für den optimalen Fall, nämlich 4 Vor- und 4 Nachglättungen fällt das Ergebnis weniger dramatisch aus, hier liegt die Konvergenzrate bei 0.107. Trotzdem bleibt die Performance weit unter den 0.050, die bei dieser Anzahl an Vor- und Nachglättungsschritten für das glatte Problem erreicht werden.

Vor- glätt.	Nach- glätt.	L gs_rb Konv.R	L gs_lex Konv.R	glatt gs_rb Konv.R	glatt gs_lex Konv.R
0	1	0.547	0.491	0.362	0.399
0	2	0.353	0.332	0.168	0.205
0	3	0.268	0.257	0.123	0.138
0	4	0.220	0.213	0.094	0.103
1	0	0.507	0.491	0.358	0.399
1	1	0.278	0.296	0.116	0.208
1	2	0.203	0.223	0.078	0.137
1	3	0.165	0.184	0.063	0.103
1	4	0.142	0.158	0.051	0.082
2	0	0.326	0.332	0.159	0.215
2	1	0.203	0.223	0.078	0.135
2	2	0.160	0.179	0.059	0.102
2	3	0.135	0.152	0.051	0.082
2	4	0.119	0.134	0.043	0.068
3	0	0.252	0.257	0.106	0.146
3	1	0.167	0.184	0.058	0.102
3	2	0.135	0.152	0.048	0.082
3	3	0.117	0.133	0.043	0.069
3	4	0.104	0.118	0.037	0.058
4	0	0.209	0.213	0.078	0.106
4	1	0.144	0.158	0.047	0.088
4	2	0.120	0.134	0.040	0.068
4	3	0.104	0.118	0.037	0.058
4	4	0.094	0.107	0.032	0.051

Tabelle 7.6: Vergleich der Konvergenzraten der Standardglätter zwischen glattem und L-Gebiet-Problem.

7.2.3 Bestimmung der Residuenreduktion für das glatte Problem

Da der Patch-adaptive Glätter jede Konvergenzrate erreichen kann, muß nun in einem weiteren Test herausgefunden werden, welche Konvergenzrate für das L-Gebiet sinnvollerweise ausgewählt werden soll. Hierbei ist zu bedenken, daß die Konvergenz eines Mehrgitterverfahrens abhängig ist von den verwendeten Komponenten. Diese sind zum einen der Glätter, zum anderen die Grobgitterkorrektur. Im L-Gebiet-Problem bricht vor allem die Performanz der Grobgitterkorrektur ein. Unser Patch-adaptiver Glätter soll nun mehr Arbeit leisten, um die schlechte Grobgitterkorrektur auszugleichen und um somit eine ähnliche Konvergenz wie im glatten Modellproblem zu erreichen.

Um dies realisieren zu können, betrachten wir das glatte (Konvergenz-) Problem genauer. Wir ermitteln die Residuen jeweils vor dem Vorglätten (r_1), nach dem Vorglätten (r_2), vor dem Nachglätten – also nach der Grobgitterkorrektur – (r_3) und nach dem Nachglätten am Ende des V-Zyklus (r_4). Die Residuenreduktion sieht also – auf allen Leveln außer dem gröbsten Level – so aus:

$$r_1 \xrightarrow{\text{Vorglätten}} r_2 \xRightarrow{\text{Grobgitterkorrektur}} r_3 \xrightarrow{\text{Nachglätten}} r_4$$

Damit läßt sich die Effizienz des Vorglätters (F_1) genauso berechnen wie die Arbeit F_2, die Grobgitterkorrektur und Nachlättung gemeinsam leisten :

$$F_1 = \tfrac{r_2}{r_1} = \tfrac{r_2}{1.0} = r_2 \quad (7.1)$$

$$F_2 = \tfrac{r_4}{r_2} \quad (7.2)$$

Aus diesen Faktoren berechnet sich die Konvergenzrate wie folgt:

$$\text{Konvergenzrate} = F_1 \cdot F_2 = \frac{r_2 \cdot r_4}{r_2} = r_4 \quad (7.3)$$

Diese Werte messen wir nach (ungefähr) 100 V-Zyklen um eine asymptotische Residuenreduktion zu erhalten. Hat sich das Problem stabilisiert, werden in jedem Schritt die gleichen Residuenreduktionen für Vor- und Nachglättung und Grobgitterkorrektur erreicht, da immer wieder dasselbe Problem berechnet wird. Da im Konvergenzproblem (Modellproblem Abschnitt 5.2) r_1 immer künstlich auf 1.0 gesetzt wurde, ist außerdem r_4 mit der Konvergenzrate identisch.

In einem ersten Versuch maßen wir die Residuenreduktion für V(2,2)-Zyklen, da 2 Vor- bzw. Nachglättungsschritte häufig das optimale Verhältnis von Glättung und Grobgitterkorrektur liefern. Da wir – wegen seiner Symmetrieeigenschaften – zum Vergleich des PAS einen rot-schwarz Glätter verwenden wollten, waren diese V(2,2)-Zyklen nicht optimal. Hier zeigte sich, daß das Problem sehr glatt ist, also wenig hochfrequente Fehleranteile enthält, die durch ein Vorglätten verbessert werden könnten. Im Gegenteil, der Rot-Schwarz-Vorglätter verschlechtert das Residuum in der Maximumsnorm sogar, da er den Fehleranteil erst auf alle schwarzen, dann auf alle roten Punkte verteilt.

Aus diesem Grund entschlossen wir uns dazu, V(0,4)-Zyklen zu betrachten, da hier auch die Entscheidung entfiel, ob die Mehrarbeit des Glätters nur während der Nachglättung oder anteilig auch schon bei der Vorglättung geleistet werden sollte.

Beachte: Für den Fall $\sharp$Vorglättungen = 0 gilt:

$$\begin{array}{lll} r_2 = & r_1 & \text{und somit} \\ F_1 = & 1.0 & \text{und weiter} \\ F_2 = & \frac{r_4}{r_2} = \frac{r_4}{r_1} = \frac{r_4}{1.0} = r_4 & \\ F_2 = & \text{Konvergenzrate} & \end{array}$$

Die Ergebnisse für beide Fälle – also V(2,2) und V(4,0) – finden sich in den Tabellen 7.7 und 7.8.

Rot-Schwarz Glätter

Level	Faktor 1	Faktor 2	Konvergenzrate
0	- -	- -	- -
1	0.597554	0.0625	0.037
2	1.20749	0.0656561	0.079
3	1.54541	0.0657923	0.102
4	1.81218	0.0641871	0.116
5	1.92281	0.0633572	0.122
6	1.97151	0.063029	0.124
7	1.99084	0.062883	0.125
8	1.99737	0.0628279	0.125
9	0.999594	0.0628102	0.063

Lexikographischer GS

Level	Faktor 1	Faktor 2	Konvergenzrate
0	- -	- -	- -
1	0.0775511	0.0603446	0.0047
2	0.302379	0.120433	0.036
3	0.308976	0.158355	0.049
4	0.360789	0.142411	0.051
5	0.591361	0.135898	0.080
6	0.838362	0.109241	0.092
7	0.955853	0.0959617	0.092
8	0.987219	0.0929952	0.092
9	0.991386	0.0926581	0.092

Tabelle 7.7: Residuenreduktion im glatten Problem für einen V(2,2)-Zyklus.

7.2.4 Die Performance des Patch-adaptiven Glätters

Nachdem sowohl die Residuenreduktion als auch die Konvergenzraten bestimmt waren, ging es an den eigentlichen Test. Hier wurde das Mehrgitterverfahren in `mg.C` so verändert, daß zuerst 100 V-Zyklen mit einem Standardglätter auf dem L-Gebiet-Konvergenzproblem gerechnet wurden. Anschließend sieht das Zwischenergebnis so aus, wie unter Abbildung 5.4

Rot-Schwarz Glätter

Level	Faktor 1	Faktor 2 = Konvergenzrate
0	- -	- -
1	1	0.060
2	1	0.152
3	1	0.178
4	1	0.184
5	1	0.185
6	1	0.186
7	1	0.186
8	1	0.186
9	1	0.093

Tabelle 7.8: Residuenreduktion im glatten Problem für einen V(0,4)-Zyklus.

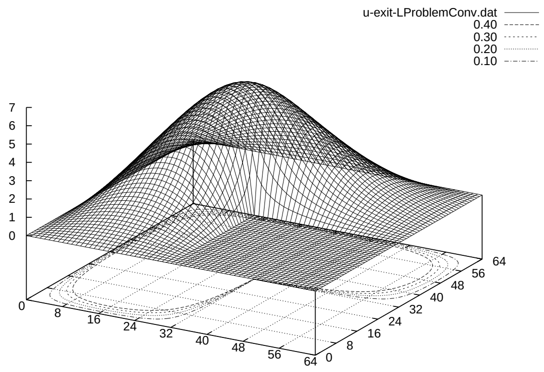
gezeigt. Dieses Zwischenergebnis wird nun als Startnäherung für einen V-Zyklus mit adaptivem Glätter verwendet, dessen Toleranz τ für jeden Level auf $0.09 \cdot \mathrm{res}_{current}$ gesetzt wurde. Nachdem dies auch für den feinsten Level der Fall ist, ist die Konvergenzrate des Zyklus somit nach oben durch 0.09 beschränkt (siehe auch Algorithmus 7.2). Damit liegt die Konvergenzrate ungefähr im Bereich der Konvergenz im glatten Problem. Nun gilt es die Anzahl der Punktrelaxationen zu messen und mit der benötigten Anzahl im Standardfall in Relation zu setzten.

Abbildung 7.4 zeigt links einen Plot des Vektors u nach 100 V-Zyklen des Mehrgitters mit Rot-Schwarz-Glätter und „artificial scaling“ (Algorithmus 7.2 Zeile 3). Die Abbildung zeigt also den Eigenvektor zum betragsgrößten Eigenwert der zu diesem Verfahren gehörenden Iterationsmatrix. Die Abbildung rechts zeigt u nach dem einen V-Zyklus mit Patch-adaptiven Glätter (Algorithmus 7.2 Zeile 9). Da die exakte Lösung des Problems der Nullvektor ist, sieht man an den absoluten Werten sehr gut, wie stark die Lösung durch den letzten V-Zyklus mit PAS verbessert wird.

Algorithmus 7.2 `mg.PASPerformance.cc`

1: **for** $iter = 0$ to `NUM_ITER` **do**
2:     scale $u^{(k)}$, so that $\mathrm{res}_{\text{finest Level}} = 1.0$ // *artificial scaling*
3:     `CS_V_cycle_STD` // *V-Cycle with standard redblack (post-)smoother*
4: **end for**
5: determine residuals for all levels
6: **for all** levels **do**
7:     $\tau^{level} = 0.09 \cdot \mathrm{res}^{level}_{current}$
8: **end for**
9: `CS_V_cycle_PAS` // *V-Cycle with PAS as (post-)smoother*
10: compute and output $\mathrm{res}_{finestlevel}$

Um die Ergebnisse dieses Tests nachvollziehbar zu machen, befindet sich auf der CD



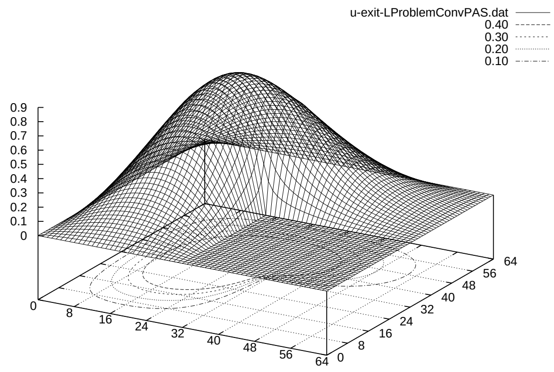


Abbildung 7.4: Die Lösung vor und nach dem `CS_V_cycle_PAS`.

im Verzeichnis `/Prog/L_Problem/ConvergenceTest/` außer `mg.STANDARD.cc` auch die Datei `mg.PAS_PERFORMANCE.cc`. Nach dem Befehl „`ln -s mg.PAS_PERFORMANCE.cc mg.C`“ kann der hier beschriebene Test durchgeführt werden.

Für diesen Test ergeben sich wirklich sehr gute Ergebnisse: Die Konvergenzrate des Standardglätters liegt bei nur 0.22 im Vergleich zu den 0.063 (< 0.09), die der Patch-adaptive Glätter erzielt. Um dieses Ergebnis erreichen zu können, braucht der Patch-adaptive Glätter auf den Levels 5-9 insgesamt 249748 Punktrelaxationen. Auf den Levels 0-4 arbeitet aufgrund der kleinen Gittergröße ein rot-schwarzer Gauß-Seidel mit 0 Vorglättungsschritten und 4 Nachglättungrelaxationen. Das ergibt für die gesamte Gitterhierarchie:

$$\begin{aligned}\sharp(\text{Punktrelax.})_{\texttt{PAS}} &= 249748 + 4 \cdot (\tfrac{3}{4} \cdot (31 \times 31 + 15 \times 15 + 7 \times 7 + 3 \times 3 + 1 \times 1)) \\ &= 249748 + 3 \cdot 1245 \\ &= 249748 + 3735 \\ &= 253483\end{aligned}$$

Der Standardglätter hingegen benötigt insgesamt:

$$\begin{aligned}\sharp(\text{Punktrelax.})_{\texttt{RB}} &= 4 \cdot (\tfrac{3}{4} \cdot (1023 \times 1023 + 511 \times 511 + 255 \times 255 + 127 \times 127 + \\ &\quad 63 \times 63 + 31 \times 31 + 15 \times 15 + 7 \times 7 + 3 \times 3 + 1 \times 1)) \\ &= 3 \cdot 1394018 \\ &= 4182054\end{aligned}$$

Ein V-Zyklus mit PAS braucht demnach weniger als $\frac{1}{10}$ der Punktrelaxationen, um auf ein weitaus besseres Ergebnis als mit dem Standardglätter zu kommen!

Kapitel 8

Schlußbemerkung

In der vorliegenden Arbeit ging es darum einen Patch-adaptiven Glätter für Mehrgitterverfahren zu entwickeln. Dazu griff man auf das bereits vorhandene Verfahren zur Punkt-adaptiven Relaxation zurück und erweiterte es auf Patches. Um nun die Vorteile eines solchen Glätters gegenüber Standardglättern zu zeigen, schließen sich an die Implementierungsphase einige Testreihen an. Sie zeigten die eindeutige Überlegenheit des implementierten Glätters für das L-Gebiet-Problem. Hier benötigte der Glätter nicht nur weniger Punktrelaxationen sondern erreichte mit diesen auch eine bessere Konvergenzrate. Es ist zu erwarten, daß dieses Ergebnis auch auf andere Modellprobleme übertragen werden kann. Dies bleibt allerdings weiterführenden Arbeiten vorenthalten.

Zusätzlich muß in anschließenden Arbeiten geklärt werden, wie viel Rechenzeitersparnis durch die Verwendung des Patch-adaptiven Glätters erreicht werden kann. Der adaptive Glätter muß durch die Verwaltung der aktiven Menge und die subtile Bestimmung der zu aktivierenden Nachbarn mehr Arbeit verrichten als ein Standardglätter, der lediglich einfache Punktrelaxationen in einer bestimmten Reihenfolge durchführt. Die bisher gewonnen Ergebnisse lassen hoffen, daß der benötigte Overhead durch die stark verringerte Anzahl von Punktrelaxationen ausgeglichen werden kann.

Auch die Zeitersparnis, die sich durch das mehrmalige Glätten eines Patches im Cache ergiebt, sollte genau bestimmt werden. Dafür ist der Umbau des Programms notwendig. Hier sollte man eine auf Cache- beziehungsweise Hardware-optimierte Version des Codes implementieren um diese dann mittels Profiling-Tools genau untersuchen zu können.

Verzeichnis der verwendeten Algorithmen

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 3. edition, 1999. `http://www.netlib.org/lapack/lug`.

[B+00] W. L. Briggs et al. *A Multigrid Tutorial.* SIAM, Philadelphia, 2000.

[Dau01] V. Daum. Runtime-controlled adaptivity in multigrid methods. Studienarbeit an der Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001.

[Han98] J. Handy. *The Cache Memory Book.* Academic Press, 2. edition, 1998.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publisher, Inc., San Francisco, California, USA, 3. edition, 2003.

[HS02] T. Huckle and S. Schneider. *Numerik für Informatiker.* Springer, 2002.

[KW03] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, March 2003.

[Lö96] H. Lötzbeyer. Parallele adaptive Mehrgitterverfahren. Master's thesis, Technische Universität München, 1996.

[Rü93] U. Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods.* Number 13 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 1993.

[TOS01] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid.* Academic Press, 2001.