Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 222

Schnelle parallele Mehrgitterlöser auf kartesischen Gittern

David Hardes

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. habil. Miriam Mehl
Betreuer/in:	Dr. rer. nat. Stefan Zimmer
Beginn am:	27. April 2015
Beendet am:	27. Oktober 2015
CR-Nummer:	G.1.8

Kurzfassung

Mehrgitterverfahren sind verbreitete Verfahren, die dazu dienen, die Lösungsapproximation eines linearen Gleichungssystemes schnell und effizient zu berechnen. Um die Verfahren durch Parallelisierung zu beschleunigen, muss man kommunikationsaufwändige Verfahren zur Glättung nutzen. Eine interessante Alternative dazu stellen die additiven Mehrgitterverfahren dar, die nicht versuchen, die verwendeten Operationen zu parallelisieren, sondern stattdessen alle Stufen der Gitterhierarchie parallel bearbeiten. Im Rahmen dieser Arbeit werden verschiedene Verfahren, unter anderem ein additives Mehrgitterverfahren, implementiert und im Hinblick auf Konvergenzeigenschaften, Speicher- und Laufzeitbedarf untersucht.

Inhaltsverzeichnis

1 Einleitung

Wenn man die physikalischen Prozesse unserer Welt berechnen möchte, bei denen man die Veränderung einer Größe in Bezug auf mehrere voneinander unabhängige Variablen betrachtet, sind partielle Differentialgleichungen eines der wichtigsten Hilfsmittel, um den realen Prozess auf ein mathematisches Modell abzubilden. Für Natur-, Ingenieurs- und Wirtschaftswissenschaften stellen sie damit ein alltäglich zu lösendes Problem dar. Das Lösen von partiellen Differentialgleichungen ist nicht trivial, sodass eine analytische Lösung des Problems meistens nicht möglich ist. Durch Diskretisierung kann man aus einer partiellen Differentialgleichung ein lineares Gleichungssystem aufstellen und das Problem somit approximativ lösen. Zur Lösung linearer Gleichungssysteme gibt es verschiedenste numerische Verfahren. Diese Arbeit beschränkt sich auf iterative Verfahren, deren Ergebnis durch Wiederholung der Verfahren die korrekte Lösung immer weiter approximiert.

Eines der einfachsten dieser Iterationsverfahren ist das Gauß-Seidel-Verfahren, welches jedoch sehr langsam konvergiert. Das sogenannte Mehrgitterverfahren konvergiert deutlich schneller und soll hier als Vergleichsbasis für ein neues Verfahren, das sogenannte additive Mehrgitterverfahren, dienen.

Im Rahmen dieser Bachelorarbeit wird eine Implementierung der Verfahren so vorgenommen, dass alle wichtigen Größen einfach ausgelesen werden können, sodass eine genaue Analyse sowie ein abschließender Vergleich dieser Verfahren vorgenommen werden kann.

Ziel dieser Arbeit ist ein Vergleich der genannten Verfahren im Hinblick auf Konvergenzverhalten sowie auf Speicherbedarf und Laufzeit.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Ausgangsproblem: definiert das Ausgangsproblem

Kapitel 3 – Lösungsverfahren: stellt die im Rahmen dieser Arbeit implementierten Verfahren vor

Kapitel 4 – Implementierung: stellt die wichtigsten Funktionen der Implementierung dar.

Kapitel 5 – Ergebnisse: enthält die Aufbereitung und Analyse der Ergebnisse der Implementierung.

Kapitel 6 – Zusammenfassung und Ausblick: fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Ausgangsproblem

Im Rahmen dieser Arbeit werden verschiedene iterative Verfahren zur numerischen Behandlung elliptischer partieller Differentialgleichungen verglichen. Alle im Laufe dieser Arbeit vorgestellten Verfahren werden dabei anhand eines typischen Modellproblems vorgestellt. Dabei wird das einfachste nichttriviale Beispiel gewählt, die Poisson-Gleichung $-\Delta u = f$.

Diskretisiert man die Differentialgleichung auf dem Einheitsquadrat $\Omega = (0,1) \times (0,1)$ mit der Maschenweite $h = \frac{1}{n}$, so ergibt sich dabei ein $n \times n$-Gitter mit n^2 Gitterpunkten. Diese Werte dieser Punkte werden mit $u_{i,j}$ bezeichnet und mit $u(x,y)) = u(i \cdot h, j \cdot h)$ berechnet, wobei die Randpunkte von Ω die Werte $u|_\Gamma = u_\Gamma$ haben. Die Punkte $u_{i,j}$ für $i,j = 1, ..., n-2$ bezeichnet man als innere Gitterpunkte. Daraus kann folgendes lineares Gleichungssystem aufgestellt werden:

$$Au = f \tag{2.1}$$

mit den Unbekannten

$$u = \begin{pmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \\ u_{1,0} & u_{1,1} & \cdots & u_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n-1,0} & u_{n-1,1} & \cdots & u_{n-1,n-1} \end{pmatrix}$$

wobei die Werte der Randpunkte $u_{i,j}$ für $i = 0$, $i = n-1$, $j = 0$ und $j = n-1$ durch die Randbedingungen vorgegeben sind, und den Konstanten

$$f = \begin{pmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,n-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ f_{n-1,0} & f_{n-1,1} & \cdots & f_{n-1,n-1} \end{pmatrix}$$

A lässt sich zerlegen in $A = D - E - F$, wobei

- D der Diagonalanteil von A,
- E die strikte untere Dreiecksmatrix,

- F die strikte obere Dreiecksmatrix ist

Zur Lösung dieses linearen Gleichungssystems gibt es nun verschiedene Möglichkeiten [Meh15].

2.1 Direkte Lösungsverfahren

Während iterative Verfahren immer nur eine Annäherung an die Lösung bieten, lösen direkte Verfahren, wie die Gauß-Elimination, immer exakt. Da der Rechenaufwand gegenüber den iterativen Verfahren jedoch deutlich größer ist, und man zudem durch die Diskretisierung der Differentialgleichung bereits einen Fehler in dem linearen Gleichungssystem hat, eine exakte Lösung demnach sowieso nicht möglich ist, werden die iterativen Verfahren in diesem Fall als Löser bevorzugt.

2.2 Iterative Lösungsverfahren

Das obige lineare Gleichungssystem 2.1 des Modellproblems lässt sich mit iterativen Verfahren relativ effizient näherungsweise lösen, wobei die Qualität der Lösung von der Feinheit des Gitters abhängt; eine feinere Abtastung ermöglicht eine genauere Diskretisierung und damit ein Ergebnis, das die Lösung besser Approximiert. Das Problem dabei ist, dass üblicherweise der Aufwand für einen Schritt des iterativen Verfahrens mit steigender Anzahl an Gitterpunkten deutlich steigt, sodass ein Optimum zwischen Genauigkeit und Geschwindigkeit gefunden werden muss. Bei manchen Verfahren hängt die Konvergenzrate zudem an der Feinheit des Gitters. Eine Verdopplung der Gitterpunkte kann eine deutlich Abnahme der Konvergenzgeschwindigkeit bedeuten. Iterative Verfahren starten üblicherweise mit einem zufällig gewählten Startvektor u_0, der in jedem Iterationsschritt um einen Fehleranteil korrigiert wird, wodurch pro Iteration eine neue Lösungsannäherung entsteht:

$$u^0 \mapsto u^1 \mapsto u^2 \mapsto u^3 \mapsto \ldots \mapsto u^m \mapsto u^{m+1} \mapsto \ldots \tag{2.2}$$

Für die Lösungsverfahren wird, so nicht anders angegeben, der Finite-Elemente-Stern verwendet, der folgendermaßen aussieht:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.3}$$

Im Folgenden werden verschiedene iterative Verfahren vorgestellt, implementiert und anschließend analysiert.

3 Lösungsverfahren

3.1 Gauß-Seidel-Verfahren

Eines der einfachsten Iterationsverfahren ist das Gauß-Seidel-Verfahren. Dabei wird für jeden Gitterpunkt in jedem Iterationsschritt über das gesamte Gitter iteriert, wobei jeder Gitterwert um einen Fehleranteil korrigiert wird, sodass sich die Gitterwerte nach und nach der Lösung annähern. Ausgehend vom zufällig gewählten Startgitter u_0, wird dabei eine Folge von korrigierten Gittern u_i berechnet[Hac91]. Dabei wird für jeden Gitterpunkt die Gleichung

$$f_{i,j} = 8 \cdot u_{i,j} - u_{i-1,j-1} - u_{i-1,j} - u_{i-1,j+1} - u_{i,j-1} - u_{i,j+1} - u_{i+1,j-1} - u_{i+1,j} - u_{i+1,j+1} \tag{3.1}$$

für jede Unbekannte $u_{i,j}$ mit $i, j := 1, ..., n-1$ gelöst. Die Randwerte des Gitters werden zwar für die Berechnung verwendet, sind als Randbedingungen jedoch auf einen Wert festgelegt. Der Fehler, sprich die Abweichung des aktuellen Wertes von der Lösung, lässt sich reduzieren, indem man für jeden Punkt das Gauß-Seidel-Verfahren als Glätter anwendet:

$$u_{i,j} = 1/8 \cdot (f_{i,j} - u_{i-1,j-1} - u_{i-1,j} - u_{i-1,j+1} - u_{i,j-1} - u_{i,j+1} - u_{i+1,j-1} - u_{i+1,j} - u_{i+1,j+1}) \tag{3.2}$$

Algorithmus 3.1 zeigt das Gauß-Seidel-Verfahren.

Algorithmus 3.1 Gauß-Seidel-Verfahren

```
procedure GS(u,f)
    while iterator < iterations do
        for k = 1 to n do
            for k = 1 to n do
                u_{i,j} = 1/8 · (f_{i,j} − u_{i−1,j−1} − u_{i−1,j} − u_{i−1,j+1} − u_{i,j−1}
                                −u_{i,j+1} − u_{i+1,j−1} − u_{i+1,j} − u_{i+1,j+1})
            end for
        end for
        iterator++
    end while
end procedure
```

Das Gauß-Seidel-Verfahren arbeitet, im Gegensatz zu ähnlichen Verfahren wie etwa dem Jacobi-Verfahren, *in-place*. Das bedeutet, dass die entsprechenden Gitterwerte bereits innerhalb eines Iterationsschrittes aktualisiert werden, wodurch der Speicherbedarf geringer ist. Dies gilt allerdings nicht für die Normalformen des Gauß-Seidel-Verfahrens, die eine modularisierte Variante des Gauß-Seidel-Verfahrens erlauben:

$$u^{m+1} = M^{GS} u^m + N^{GS} f \tag{3.3}$$

mit

- $M^{GS} = (D - E)^{-1} F$
- $N^{GS} = (D - E)^{-1}$

Das Gauß-Seidel-Verfahren aktualisiert also pro Iteration nur die direkt benachbarten Punkte um die Information aus einem Punkt. Daher benötigt man im ungünstigsten Fall n Iterationen, um zum Beispiel Punkt $u_{1,1}$ um den Wert aus Punkt $u_{n-1,n-1}$ zu aktualisieren.

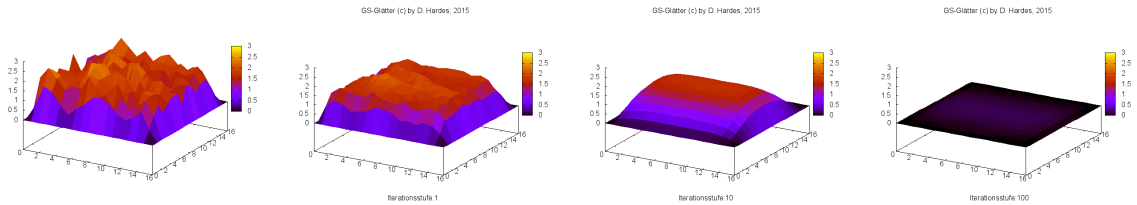

Abbildung 3.1: Gauß-Seidel angewendet auf ein 17×17-Gitter
von links: Startvektor, 1 Iteration, 10 Iterationen, 100 Iterationen

Betrachtet man Abbildung 3.1, die den Zustand des Gitters u^0, u^1, u^{10} und u^{100} des linearen Gleichungssystems $Au = 0$ mit den Randwerten $u|_\Gamma = 0$ nach 0,1,10 und 100 Iterationen des Gauß-Seidel-Verfahrens darstellt, so fällt auf, dass das Verfahren in den ersten paar Iterationsschritten den Fehler noch schnell korrigiert, danach jedoch stark abfällt und den Fehler nur noch langsam reduziert. Dies liegt darin begründet, dass sich der Fehler über das ganze Gitter gesehen in hoch- und niederfrequente Anteile relativ zur Anzahl der Gitterpunkte zerlegen lässt. Die hochfrequenten Anteile lassen sich leicht auf dem Gitter glätten, man sieht jedoch auch, dass der niederfrequente Anteil sich nur schlecht glätten lässt, da hier in jedem Iterationsschritt nur ein minimaler Anteil korrigiert werden kann. Der Grund für dieses Verhalten liegt in Formel 3.2: Es werden nur die unmittelbar benachbarten Punkte zur Berechnung des aktualisierten Wertes herangezogen. Zur Korrektur des hochfrequenten Fehleranteil ist dies kein Problem, da zwischen den Werten zweier benachbarter Gitterpunkte eine große Differenz liegt. Die Korrektur des niederfrequenten Anteils hingegen ist deutlich iterationsaufwendiger, da die Differenz zwischen den Werten zweier benachbarter Gitterpunkte nicht groß ist.

Daran zeigt sich, dass das Verfahren, obwohl es letztendlich gegen die Lösung konvergiert, dieses zu langsam macht, als dass man es sinnvoll einsetzen könnte. Es wird sich jedoch zeigen, dass das Gauß-Seidel-Verfahren gut als Glätter für die in den folgenden Kapiteln beschriebenen Mehrgitterverfahren geeignet ist. Dabei wird zuerst das klassische Mehrgitterverfahren nach Hackbusch [Hac91] vorgestellt, das, bewiesen und bekannt, als Kontrollverfahren für das neue und weniger erforschte additive Mehrgitterverfahren nach [VY14] dienen soll.

3.2 Mehrgitterverfahren

Mehrgitterverfahren gehören zu den am schnellsten konvergierenden iterativen Verfahren. Wie in Kapitel 3.1 beschrieben, gibt es verschiedene Fehleranteile mit unterschiedlichen Frequenzen, womit das Gauß-Seidel-Verfahren nicht gut zurechtkommt. Wenn man die Fehleranteile jedoch genauer betrachtet, fällt auf, dass die Frequenz der Fehler immer relativ zur Anzahl der Gitterpunkte ist. Wenn man also weniger Gitterpunkte betrachtet, ist der Fehler niederfrequenter und damit leichter zu korrigieren.

Diese Möglichkeit macht man sich bei den Mehrgitterverfahren zunutze, indem man den Fehler auf unterschiedlich fein aufgelösten Gittern korrigiert.

Die Mehrgitterverfahren nutzen eine Hierarchie von Gleichungssystemen, bei denen auf jeder Ebene ein lineares Gleichungssystem der Form $A_\ell x_\ell = b_\ell$ gelöst werden muss. Wenn das Ausgangsgitter der Stufe ℓ, also x_ℓ (und dementsprechend auch das dazugehörende Konstantengitter b_ℓ) $n \times n$ Gitterpunkte hat, so sollen die beiden Gitter der nächstgröberen Hierarchiestufe $\ell - 1$, also $x_{\ell-1}$ und $b_{\ell-1}$, noch $\frac{n}{2} \times \frac{n}{2}$ Gitterpunkte haben.

Dieses gilt sinngemäß auch für die nächsten Stufen der Hierarchie von Gleichungssystemen, bis hin zur Gitterstufe $\ell = 0$, bei der nur noch 1 innerer Gitterpunkt, umgeben von 8 Randpunkten, vorhanden ist.

Um aus dem Gleichungssystem der Stufe ℓ das Gleichungssystem der Stufe $\ell - 1$ abzuleiten, benötigen wir für jede Gitterstufe

- eine **Prolongation**, die die Korrekturwerte von einem groben auf ein feines Gitter überträgt,
- eine **Restriktion** die das Residuum von einem feinen auf ein grobes Gitter überträgt,
- und einen **Glätter**, der eine Lösungsapproximation für die aktuelle Stufe liefert.

Zudem benötigt man eine Möglichkeit, das Residuum zu Berechnen, das durch die Restriktion restringiert werden soll, sowie eine Grobgitterkorrektur, die die aktuelle Lösung um die prolongierten Werte der gröberen Stufe korrigiert. Diese Operatoren werden im folgenden Abschnitt beschrieben.

3.2.1 Glätter

Der Glätter hat beim Mehrgitterverfahren die Aufgabe, für jede Gitterstufe ℓ eine Lösungsapproximation zu liefern. Da wir das Mehrgitterverfahren nutzen wollen, um das Problem der niederfrequenten Fehleranteile zu lösen, brauchen wir ein Lösungsverfahren, dass sich in erster Linie um den hochfrequenten Fehleranteil kümmert. Wie wir in Kapitel 3 gesehen haben, ist das Gauß-Seidel-Verfahren ein ebensolcher Löser. Eine Iteration des GS-Verfahrens ist relativ günstig und korrigiert dabei bereits einen großen Anteil des Fehlers. Für eine erste Version kann man das GS-Verfahren sowohl als Vor- und Nachglätter, als auch als Direktlöser für die feinste Stufe verwenden.

Das in Kapitel 3.3 beschriebene additive Mehrgitterverfahren nutzt allerdings für die Glättung eine zerlegt Form des Gauss-Seidel-Verfahrens. Um eine Vergleichbarkeit der Verfahren herstellen zu können, wird das multiplikative Mehrgitterverfahren als Nachglätter den sogenannten Rückwärts-Gauß-Seidel nutzen, bei dem die Iteration über das Gitter einfach in umgekehrter Reihenfolge stattfindet, sprich in Gleichung 3.2 wird nicht, wie bisher, über $i, j = 1, ..., n-1$ iteriert, sondern über $i, j = n-1, ..., 1$. Da für das Mehrgitterverfahren verschiedene Glätter möglich sind, stellt dies für die Konvergenzeigenschaften der Mehrgitterverfahren keine Einschränkung dar.

3.2.2 Prolongation

Um die Werte des groben Gitters auf das feine Gitter zu übertragen, benötigt man einen Operator, der eine lineare Abbildung vom groben auf das feine Gitter ermöglicht, und zwar die sogenannte Prolongation, definiert als eine lineare Abbildung vom groben aufs feine Gitter. Dabei bietet sich der Einfachheit halber an, als Prolongation stückweise lineare Interpolation zwischen den Gitterpunkten des groben Gitters zu verwenden. Die Prolongation P ist definiert als

$$P_{\ell-1}^{\ell} : u_{\ell-1} \to u_{\ell} \tag{3.4}$$

wobei P den Stern

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix} \tag{3.5}$$

und die sogenannte Neunpunktprolongation darstellt. Daraus ergibt sich eine Reihe von Gleichungen mit denen man über das Gitter traversiert:

$$u^{\ell+1}_{2i,2j} = u^{\ell}_{i,j} \tag{3.6}$$

$$u^{\ell+1}_{2i+1,2j} = 1/2(u^{\ell}_{i+1,j} + u^{\ell}_{i,j}) \tag{3.7}$$

$$u^{\ell+1}_{2i,2j+1} = 1/2(u^{\ell}_{i,j+1} + u^{\ell}_{i,j}) \tag{3.8}$$

$$u^{\ell+1}_{2i-1,2j} = 1/2(u^{\ell}_{i-1,j} + u^{\ell}_{i,j}) \tag{3.9}$$

$$u^{\ell+1}_{2i,2j-1} = 1/2(u^{\ell}_{i,j-1} + u_{i,j^{\ell}}) \tag{3.10}$$

$$u^{\ell+1}_{2i+1,2j+1} = 1/4(u^{\ell}_{i+1,j} + u^{\ell}_{i,j+1} + u^{\ell}_{i+1,j-1} + u^{\ell}_{i,j}) \tag{3.11}$$

$$u^{\ell+1}_{2i-1,2j-1} = 1/4(u_{i-1,j^{\ell}} + u^{\ell}_{i,j-1} + u^{\ell}_{i-1,j+1} + u^{\ell}_{i,j}) \tag{3.12}$$

$$u^{\ell+1}_{2i+1,2j-1} = 1/4(u^{\ell}_{i+1,j} + u^{\ell}_{i,j-1} + u^{\ell}_{i+1,j-1} + u^{\ell}_{i,j}) \tag{3.13}$$

$$u^{\ell+1}_{2i-1,2j+1} = 1/4(u^{\ell}_{i-1,j} + u^{\ell}_{i,j+1} + u^{\ell}_{i-1,j+1} + u^{\ell}_{i,j}) \tag{3.14}$$

3.2.3 Residuumsberechnung

Um die Werte des Gitters der Stufe ℓ auf das nächstgröbere Gitter mit der Stufe $\ell + 1$ zu übertragen, benötigt man die Restriktion. Diese überträgt das Residuum von u_ℓ, und nicht u_ℓ selber auf ein gröberes Gitter. Das Residuum eines Gitters u_ℓ berechnet man durch $f_\ell - A_\ell u_\ell$, wodurch für jedes $u_{i,j}$, mit $i, j = 1, ..., n-1$ gilt:

$$\begin{aligned} r^{\ell}_{i,j} = f^{\ell}_{i,j} - (8 \cdot u^{\ell}_{i,j} &- u^{\ell}_{i-1,j-1} - u^{\ell}_{i-1,j} \\ &- u^{\ell}_{i-1,j+1} - u^{\ell}_{i,j-1} \\ &- u^{\ell}_{i,j+1} - u^{\ell}_{i+1,j-1} \\ &- u^{\ell}_{i+1,j} - u^{\ell}_{i+1,j+1}) \end{aligned} \tag{3.15}$$

3.2.4 Restriktion

Die Restriktion ist definiert als eine lineare Abbildung vom feinen Gitter auf das grobe Gitter. Dabei wird als Restriktion R einfach die transponierte Prolongation verwendet. R ist definiert als

$$R^{\ell}_{\ell-1} = (P^{\ell}_{\ell-1})^T : u_\ell \to u_{\ell-1} \tag{3.16}$$

wobei R den Stern

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix} \tag{3.17}$$

und damit die Neunpunktrestriktion darstellt. Daraus ergibt sich eine Gleichung, mit der man über die Gitterpunkte traversiert:

$$u_{i,j} = r_{2i,2j} + 1/2 \cdot (r_{2i-1,2j} + r_{2i+1,2j} + r_{2i,2j+1} + r_{2i,2j-1}) + 1/4 \cdot (r_{2i-1,2j-1} + r_{2_i+1,2j+1} + r_{2i+1,2j-1} + r_{2i-1,2j+1}) \tag{3.18}$$

3.2.5 Grobgitterkorrektur

Um das Eingabegitter u_m mit den Werten des Fehlergitters e_m zu korrigieren, werden die Werte der beiden Gitter komponentenweise addiert. Voraussetzung dafür ist, dass die Werte der gröberen Gitterstufen durch die Prolongation auf die feinste Gitterstufe übertragen wurden, da eine Addition ansonsten nicht möglich ist.

$$u^{m+1} = u^m + e^m \tag{3.19}$$

3.2.6 Zweigitterverfahren

Das einfachste Verfahren, das mit mehreren Gitterstufen arbeitet, ist das Zweigitterverfahren. Dabei wird die Restriktion nur einmal durchgeführt, es arbeitet also nur auf zwei Gitterstufen. Der Algorithmus 3.2 zeigt den Ablauf dieses Verfahrens:

Algorithmus 3.2 Zweigitterverfahren

procedure ZG(u_f,f_f)
  GS(u_f, f_f)
  $r_f = b_f - Au_f$
  $r_c = R_H^h r_f$
  $e_c = A_H^{-1} r_c$
  $e_f = P_H^h e_c$
  $u_f = u_f + e_f$
  GS(u_f, f_f)
end procedure

In einem Iterationsschritt des Verfahrens werden die Werte des Startgitters durch eine Iteration des Gauß-Seidel-Verfahres vorgeglättet, das Residuum des geglätteten Gitters wird auf das nächstgröbere Gitter restringiert. Auf dieser Gitterstufe wird das Korrekturgitter der Grobgitterstufe e_c durch $A_H^{-1} r_c$ berechnet, was man durch eine große Anzahl an Glätteriterationen erreichen kann. Das Startgitter wird anschließend um die prolongierte Lösungsapproximation korrigiert sowie nachgeglättet.

Sollte die Qualität der Approximation noch nicht ausreichend sein, so wiederholt man das ganze Verfahren entsprechend oft.

Dieses Verfahren konvergiert deutlich schneller als das Gauß-Seidel-Verfahren, die Laufzeit einer Iteration ist allerdings aufgrund der Vor- und Nachglättung sowie der großen Anzahl an Grobgitter-Gauß-Seidel-Iterationen deutlich schlechter.

3.2.7 Mehrgitterverfahren

Während das Zweigitterverfahren nur einmal auf ein gröberes Gitter wechselt, arbeitet das Mehrgitterverfahren auf allen möglichen Gitterstufen, d.h. die Restriktion wird durchgeführt bis nur noch ein innerer Gitterpunkt verfügbar ist. Für diesen wird das entsprechende Gleichungssystem durch Anwendung des Gauß-Seidel-Verfahrens korrekt gelöst. Statt also auf der gröberen Gitterstufe das Gauß-Seidel-Verfahren anzuwenden, führt man erneut einen Mehrgitterschritt durch. Dadurch erhält man ein rekursives Verfahren, dessen Ablauf in Algorithmus 3.3 zu sehen ist:

Algorithmus 3.3 Mehrgitterverfahren

procedure MMG(u_ℓ,f_ℓ)
  if $\ell = 0$ **then**
    GS(u_ℓ, f_ℓ)
  else
    GS(u_ℓ, f_ℓ)
    $r_\ell = b_f - Au_\ell$
    $r_{\ell-1} = R_H^h r_\ell$
    $e_{\ell-1} = \text{MMG}(e_{\ell-1}, r_{\ell-1})$
    $e_\ell = P_H^h e_{\ell-1}$
    $u_\ell = u_\ell + e_\ell$
    GS(u_ℓ, f_ℓ)
  end if
end procedure

Dieses rekursive Verfahren wird auch *V-Zyklus* genannt, da die Gitterstufen V-förmig von feinsten über bis zum gröbsten Gitter restringiert und anschließend vom gröbsten zum feinsten prolongiert werden. Auch bei diesem Verfahren kann man die Qualität der Lösungsapproxmiation durch iterative Anwendung verbessern.

Dieses Verfahren konvergiert ebenfalls schneller als das Gauß-Seidel-Verfahren und ist dabei in der Ausführung deutlich schneller als das Zweigitterverfahren, da beim Zweigitterverfahren eine große Anzahl an Gauß-Seidel-Iterationen auf der Grobgitterstufe stattfindet,

wohingegen sich beim Mehrgitterverfahren auf jeder Gitterstufe die Anzahl der zu berechnenden Werte um ¾ reduziert. Der Rechenaufwand des Mehrgitterverfahrens verhält sich dabei linear zur Anzahl der Gitterpunkte.

Dementsprechend ist die zu erwartende Laufzeit des Zweigitterverfahrens deutlich schlechter als die des Mehrgitterverfahrens.

Um eine Vergleichbarkeit mit dem im folgenden Kapitel 3.3 vorgestellten additiven Mehrgitterverfahren zu ermöglichen, wird die Nachglättung bei Vergleichen dieser beiden Verfahren durch das vorher beschriebene Rückwärts-Gauß-Seidel-Verfahren geglättet, bei dem statt von Gitterpunkt $1, 1$ nach $n-2, n-2$ zu iterieren von $n-2, n-2$ nach $1, 1$ iteriert wird.

3.3 Additives Mehrgitterverfahren

Das in Kapitel 3.2 beschriebene multiplikative Mehrgitterverfahren besitzt bereits eine hohe Konvergenzgeschwindigkeit. Die Laufzeit der Berechnungen ist jedoch dadurch beschränkt, dass die Operationen sequentiell durchgeführt werden müssen. Möchte man deutlich schneller werden, so kann man an dem Verfahren nicht mehr viel optimieren; einen messbaren Geschwindigkeitszuwachs erhält man nur noch durch Parallelisierung der Berechnungen. Leider haben die meisten Löser das Problem, dass sie sich nicht oder nur schlecht parallelisieren lassen - es gibt Verfahren, um das Mehrgitterverfahren, wie in Kapitel 3.2 vorgestellt, zu parallelisieren, indem man das zu lösende Gitter nach mehr oder minder komplexen Verfahren aufteilt, dabei entsteht allerdings ein gewaltiger Kommunikationsaufwand, was für eine parallele Implementierung nicht erwünscht ist. Statt nun horizontal zu parallelisieren, also im V-Zyklus auf einer Gitterstufe die Operationen zu parallelisieren, parallelisiert man vertikal: Man berechnet sämtliche Gitterstufen parallel [VY14]. Das additive Mehrgitterverfahren, auch BPX genannt, setzt genau dieses um. Gegenüber dem multiplikativen Mehrgitterverfahren kann es beim BPX unter Umständen zu Problemen mit der Stabilität kommen. Daher haben die Mathematiker P.S. Vassilevski und U.M. Yang ein Verfahren entwickelt, dass die Stabilität der multiplikativen Mehrgitterverfahren mit den Laufzeiteigenschaften der additiven Verfahren kombiniert. Dabei lassen sich die Operationen des bisherigen multiplikativen Mehrgitterverfahrens teilweise wiederverwenden.

3.3.1 Operatoren

Das additive Mehrgitterverfahren funktioniert grundsätzlich nach den gleichen Prinzipien wie das multiplikative Mehrgitterverfahren aus Kapitel 3.2. Dazu müssen die bekannten Operatoren leicht erweitert bzw. verändert werden.

Statt des bisher verwendeten Verfahrens wird eine Zerlegung des modularisierten Gauß-Seidel-Verfahrens verwendet, wie sie in Gleichung 3.3 beschrieben ist. Dabei wurde gezeigt,

dass das multiplikative Mehrgitterverfahren mit Rückwärts-Gauss-Seidel als Nachglätter den in Gleichung 3.20 beschriebenen Mehrgitteroperator hat, der ähnliche Eigenschaften wie der des additiven Mehrgitterverfahrens aus Gleichung 3.21 hat [VY14].

Demnach ist es möglich, statt jeweils einer Iteration des Gauß-Seidel-Verfahrens als Vor- und Nachglätter sowie auf der gröbsten Gitterstufe als Direktlöser zu verwenden, auf jeder Gitterstufe die Operation $\bar{N}$ (symmetrisierter Glätter, Formel 3.22) in Kombination mit den geglätteten Varianten der Prolongation und Restriktion zu verwenden, um das selbe Ergebnis zu erhalten.

Dementsprechend sollten beide Verfahren exakt die gleiche Lösung liefern.

$$N_\ell^{multMG} = \bar{N}_\ell + (I - N_\ell^T A_\ell) P_{\ell-1}^\ell N_{\ell-1}^{multMG} R_{\ell-1}^\ell (I - A_\ell N_\ell) \tag{3.20}$$

$$N^{addMG} = \sum_{j=0}^{\ell} (\bar{P}_j \bar{N} \bar{R}_j) \tag{3.21}$$

$$\bar{N}_\ell = N_\ell^T D_\ell N_\ell \tag{3.22}$$

Prolongation

Die bisher bekannte Prolongationsoperation wird nur um eine Glättungsoperation erweitert und ergibt damit die sogenannte geglättete Prolongation.

$$\bar{P}_{\ell-1}^\ell = (I - N_\ell^T A_\ell) P_{\ell-1}^\ell \tag{3.23}$$

Restriktion

Für die Restriktion gilt prinzipiell das gleiche wie für die Prolongation, hier muss man allerdings auch die Glättungsoperation transponieren.

$$(\bar{P}_{\ell-1}^\ell)^T = ((I - N_\ell^T A_\ell) P_{\ell-1}^\ell)^T = \bar{R}_{\ell-1}^\ell = (I - A_\ell N_\ell) R_{\ell-1}^\ell \tag{3.24}$$

3.3.2 Mehrgitteraufruf

Der Algorithmus 3.4 zeigt den Algorithmus der beschriebenen Variante des additive Mehrgitterverfahrens.

Algorithmus 3.4 additives Mehrgitterverfahren

procedure AMG(u_ℓ,f_ℓ)
  for $k = \ell, ..., 0$ **do**
    $r_\ell = b_\ell - Au_\ell$
    $r_{\ell-1} = \bar{R}_H^h r_\ell$
  end for
  for $k = \ell, ..., 0$ **do**
    $e_\ell = \bar{N} r_\ell$
  end for
  for $k = \ell, ..., 0$ **do**
    $u_\ell = u_\ell + (\bar{P}_H^h)_\ell e_{\ell-1}$
  end for
end procedure

Die zweite `for`-Schleife ist parallelisierbar, da alle notwendigen Daten durch die Restriktionsschleife, die erste `for`-Schleife, berechnet wurden. Dieser Schritt muss zwangsläufig parallel erfolgen, die Restriktion von $u_{\ell-1}$ nach $u_{\ell-2}$ kann erst passieren, wenn die Restriktion von u_ℓ nach $u_{\ell-1}$ abgeschlossen ist.

Theoretisch kann man damit so viele Rechenwerke, wie man Gitterstufen hat, auslasten. Um optimale Geschwindigkeit zu erreichen, reicht jedoch auch eine Verteilung auf mehrere Rechenwerke, da der Rechenaufwand für das feinste Gitter am höchsten ist. Für alle feineren Gitter viertelt sich die Anzahl der benötigten Operationen, sodass man diese auf weniger Rechenwerke verteilen kann und trotzdem noch eine theoretische Maximallaufzeit von t_{u^0} hat. Auch die Prolongation kann parallel erfolgen, am Ende eines Mehrgitterzyklus werden jedoch alle Korrekturgitter in der Größe der feinsten Gitterstufe vorliegen, aufeinander aufaddiert und anschließend zur Fehlerkorrektur verwendet. Daran zeigt sich schon ein potentielles Problem: Um die Lösungsapproximation von ℓ Gitterstufen abzuspeichern, benötigt man $\ell \cdot \mathtt{M}_{u^0}$, wenn man die Operationen und die Speicherverwaltung möglichst einfach halten will. Demgegenüber benötigt das multiplikative Mehrgitterverfahren nur $1,5 \cdot \mathtt{M}_{u^0}$, da hier rekursiv auf einer Datenstruktur gearbeitet wird.

Eine genaue Analyse der Speicher- und Laufzeitproblematik der Implementierung aus dem folgenden Kapitel folgt in Kapitel 5

4 Implementierung

In diesem Kapitel wird die programmiertechnische Seite der Implementierung vorgestellt. Die gesamte Arbeit wurde in `C` nach Standard `C99` geschrieben und mit `GCC` unter Linux kompiliert. Die Steuerung des Programms findet durch ein Shell-Skript statt, das einen vereinfachten Aufruf sowie eine Aufbereitung der Ergebnisse übernimmt und sich dabei auch noch um das Aufräumen der temporären Dateien kümmert.

Im Allgemeinen wurde bei der Implementierung mehr auf Speichereffizienz als auf Schnelligkeit Wert gelegt. Dies ist unter anderem an der häufigen Allokation von Hilfsgittern erkennbar; das Programm würde möglicherweise deutlich schneller durchlaufen, wenn gerade die Hilfsgitter, die häufig verwendet werden, am Anfang alloziert würden. Einerseits müssten dann deutlich mehr Pointer übergeben werden, was die Signaturen verlängern und damit die Lesbarkeit reduzieren und die Komplexität erhöhen würde. Andererseits wäre es nicht unbedingt ein Nachteil, dass das Programm für die gesamte Laufzeit einen großen Bereich des Arbeitsspeichers alloziert - so steht von Anfang an fest, dass das Programm diesen Bereich nutzen kann. Bei der dynamischen Allokation und Freigabe kann es theoretisch vorkommen, dass ein anderes Programm von einer auf die nächste Iteration den Speicher belegt - das Programm würde abstürzen. Da das Ziel der Arbeit jedoch die Implementierung eines übersichtliches Programms war, bei dem man Zugriff auf verschiedenste Parameter und Eigenschaften hat, wurden Laufzeitnachteile gegenüber Speicheroptimierung in Kauf genommen.

Der Code wurde zur einfacheren Nutzung auf mehrere Bibliotheken aufgeteilt, wobei die Trennung in Abhängigkeit der Funktionalität erfolgt. Die Bibliothek `memory.c` enthält die Speicheroperationen, `grid.c` die grundlegenden Gitteroperationen wie Addition und Skalierung. Die Mehrgitter- (`multigrid.c`) sowie die Glätterbibliothek (`smoother.c`) enthalten die eigentlichen Löser, während das Rahmenprogramm in `main.c` zu finden ist. Funktionen zur Ein- und Ausgabe sind in `io.c` eingeordnet.

4.1 Datenstruktur

Die Daten, auf denen gerechnet wird, müssen im Arbeitsspeicher abgelegt werden. Für das Poisson-Problem wird das lineare Gleichungssystem $Au = f$ gelöst. Dafür muss man zwei

Gitter vorhalten: Zum einen das Gitter u, dass die Arbeitsdaten enthält, zum anderen das Gitter f, dass die Konstanten enthält.

Da man sowohl die Randpunkte, als auch die Gitterpunkte für die Berechnung benötigt, auch wenn die Randpunkte nicht zur Lösung gehören, werden diese in einem gemeinsamen Gitter gespeichert. Für die MG-Verfahren, die auf den gröberen Gittern keine Randbedingungen mehr haben, sind diese Randwerte einfach Null.

Während man für kleine Probleminstanzen mit Gittergrößen im Bereich von 1000×1000 noch statische Arrays verwenden kann, ist dies für größere Gitterdimensionen nicht mehr möglich. Deswegen wurde eine Datenstruktur implementiert, die Gittergrößen theoretisch nur von der Größe des Arbeitsspeichers abhängig macht.

Daher werden hier Speicherbereiche statisch alloziert. Die im Quellcode abgebildete Datenstruktur stellt die notwendige Pointerstruktur bereit, mit der man einen Gitterpointer für ein Gitter der Größe |`size`| mit einem Pointer auf den Speicherbereich erzeugen kann. Dieser erleichtert den Datenzugriff, da die Werte eines Gitters einfach über den Pointer erreicht werden können.

```
1 struct _grid{
2     int size;
3     double** grid;
4 };
5 typedef struct _grid _grid_t;
6 typedef _grid_t * _grid_p;
```

`_grid_p` ist also ein Pointer auf den Speicherbereich, an dem ein `c-struct` abgelegt ist, dass aus einer Integerzahl für die Größe des Gitters und einem weiteren Pointer auf den Speicherbereich, an dem die Gitterpunkte abgelegt sind, besteht.

Zwei Funktionen stellen einfache Schnittstellen bereit, mit denen man einfach Gitter unter Angabe eines Namens und einer Größe anlegen (`allocateGrid`), und wieder freigeben (`freeGrid`) kann. Dabei initialisieren die implementierten Allokationsfunktionen grundsätzlich alle Speicherbereiche mit 0, da dies nicht automatisch durch den Compiler geschieht.

4.2 Glätter

Im Rahmen dieser Arbeit werden mehrere Glätter verwendet. Neben dem einfachen (Vorwärts- und Rrückwärts-) Gauß-Seidel-Verfahren wird auch das in seine Normalformen zerlegte Gauß-Seidel-Verfahren benötigt.

4.2.1 Gauss-Seidel-Verfahren

Im Folgenden ist die Implementierung des Vorwärts-Gauss-Seidel-Verfahrens nach Formel 3.2 abgebildet, dabei wird über alle inneren Gitterpunkte iteriert.

```
1  void simpleGaussSeidel(_grid_p u, _grid_p f){
2      int i,j;
3      for (i = 1; i < u->size-1; i++){
4          for (j = 1; j < u->size-1; j++){
5              u->grid[i][j]  = 0.125*(u->grid[i-1][j-1] + u->grid[i-1][j]
6                                    + u->grid[i-1][j+1] + u->grid[i][j-1]
7                                    + u->grid[i][j+1]   + u->grid[i+1][j-1]
8                                    + u->grid[i+1][j]   + u->grid[i+1][j+1]
9                                    + f->grid[i][j]);
10         }
11     }
12 }
```

Die für den Vergleich des multiplikativen und additiven Mehrgitterverfahrens benötigte Rückwärts-Gauß-Seidel-Variante unterscheidet sich vom Vorwärts-Gauß-Seidel nur durch die Iterationsreihenfolge; man iteriert nicht zeilenweise von $u_{1,1}$ nach $u_{n-1,n-1}$, sondern zeilenweise von $u_{n-1,n-1}$ nach $u_{1,1}$.

```
1  void simpleGaussSeidelB(_grid_p u, _grid_p f){
2      int i,j;
3      for (i = 1; i < u->size-1; i++){
4          for (j = 1; j < u->size-1; j++){
5              u->grid[i][j]  = 0.125*(u->grid[i-1][j-1] + u->grid[i-1][j]
6                                    + u->grid[i-1][j+1] + u->grid[i][j-1]
7                                    + u->grid[i][j+1]   + u->grid[i+1][j-1]
8                                    + u->grid[i+1][j]   + u->grid[i+1][j+1]
9                                    + f->grid[i][j]);
10         }
11     }
12 }
```

4.2.2 Modulares Gauss-Seidel-Verfahren

Für das additive Mehrgitterverfahren benötigt man Bestandteile des modularen Gauß-Seidel-Verfahrens. Die Zerlegung in zwei Funktionen erfolgt nach Formel 3.3. Dabei werden zwei Gitter angelegt, in die die Ergebnisse der Berechnungen $l = M^{GS}u$ sowie $r = N^{GS}f$ gespeichert werden. Diese Gitter werden in Zeile 8 einfach addiert, anschließend wird der Speicher wieder freigegeben

```
void modularGaussSeidel(_grid_p u, _grid_p f){
    int i,j;
    _grid_p l, r;
    l = allocateGrid(l, u->size);
    r = allocateGrid(r, f->size);
    mdotu(u, l);
    ndotb(f, r);
    addGrid(l, r, u);
    freeGrid(l);
    freeGrid(r);
}
```

Die Hilfsfunktion `mdotu` implementiert die Formel $l = M^{GS}u$:

```
void mdotu(_grid_p u, _grid_p l){
    int i,j;
    for (i = 1; i < u->size-1; i++){
        for (j = 1; j < u->size-1; j++){
            l->grid[i][j] = 0.125*(l->grid[i-1][j-1] + l->grid[i][j-1]
                                 + l->grid[i-1][j+1] + l->grid[i-1][j]
                                 + u->grid[i+1][j]   + u->grid[i][j+1]
                                 + u->grid[i+1][j-1] + u->grid[i+1][j+1]);
        }
    }
}
```

Die Hilfsfunktion `ndotb` implementiert die Formel $r = N^{GS}f$:

```
void ndotb(_grid_p f, _grid_p r){
    int i,j;
    for (i = 1; i < b->size-1; i++){
        for (j = 1; j < b->size-1; j++){
            r->grid[i][j] = 0.125*(r->grid[i-1][j-1] + r->grid[i][j-1]
                                 + r->grid[i-1][j+1] + r->grid[i-1][j]
                                 + f->grid[i][j]);
        }
    }
}
```

Für das additive Mehrgitterverfahren benötigt man zudem die Hilfsfunktion $(N^{GS})^T$:

```
void ndotbT(_grid_p f, _grid_p r){
    int i,j;
    for (i = b->size-2; i > 0 ; i--){
        for (j = b->size-2; j > 0; j--){
            r->grid[i][j] = 0.125*(r->grid[i+1][j-1] + r->grid[i][j+1]
                                 + r->grid[i+1][j+1] + r->grid[i+1][j]
                                 + f->grid[i][j]);
        }
    }
}
```

4.3 Mehrgitterverfahren

Für das Mehrgitterverfahren benötigt man Funktionen zur Berechnung von Residuum sowie für die Prolongation und Restriktion.

4.3.1 Prolongationsoperator

Die Prolongation als lineare Abbildung von einem groben auf ein feines Gitter wurde anhand der Formeln 3.5 bis 3.14 implementiert:

```
void prolongation(_grid_p e_c, _grid_p e_f){
    int i,j, ii, jj;
    for (i = 1; i < e_c->size-1; i++){
        for (j =1; j < e_c->size-1; j++){
            ii=(2*i);
            jj=(2*j);
            e_f->grid[ii][jj]     =       e_c->grid[i][j];
            e_f->grid[ii+1][jj]   =  0.5*(e_c->grid[i+1][j] + e_c->grid[i][j]);
            e_f->grid[ii][jj+1]   =  0.5*(e_c->grid[i][j+1] + e_c->grid[i][j]);
            e_f->grid[ii-1][jj]   =  0.5*(e_c->grid[i-1][j] + e_c->grid[i][j]);
            e_f->grid[ii][jj-1]   =  0.5*(e_c->grid[i][j-1] + e_c->grid[i][j]);
            e_f->grid[ii+1][jj+1] = 0.25*(e_c->grid[i][j]   + e_c->grid[i+1][j]
                                          + e_c->grid[i][j+1] + e_c->grid[i+1][j+1]);
            e_f->grid[ii-1][jj-1] = 0.25*(e_c->grid[i][j]   + e_c->grid[i-1][j]
                                          + e_c->grid[i][j-1] + e_c->grid[i-1][j-1]);
            e_f->grid[ii+1][jj-1] = 0.25*(e_c->grid[i][j]   + e_c->grid[i+1][j]
                                          + e_c->grid[i][j-1] + e_c->grid[i+1][j-1]);
            e_f->grid[ii-1][jj+1] = 0.25*(e_c->grid[i][j]   + e_c->grid[i-1][j]
                                          + e_c->grid[i][j+1] + e_c->grid[i-1][j+1]);
        }
    }
}
```

4.3.2 Restriktionsoperator

Die Restriktion als lineare Abbildung von einem feinen auf ein grobes Gitter wurde nach Formel 3.19 implementiert:

```
void restriction(_grid_p u_c, _grid_p r){
    int i, j, ii, jj;
    for (i = 1; i < u_c->size - 1; i++){
        for (j = 1; j < u_c->size - 1; j++){
            ii=(2*i);
            jj=(2*j);
            u_c->grid[i][j] = (r->grid[(ii)][(jj)]
```

```
                        + 0.5* (r->grid[(ii-1)][(jj)]   + r->grid[(ii+1)][(jj)]
                             +  r->grid[(ii)][(jj-1)]   + r->grid[(ii)][(jj+1)])
                        + 0.25*(r->grid[(ii-1)][(jj+1)] + r->grid[(ii+1)][(jj+1)]
                             +  r->grid[(ii+1)][(jj-1)] + r->grid[(ii-1)][(jj-1)]));
        }
    }
}
```

Das Residuum wird mithilfe der Formel $r = (f - Au)$ berechnet:

```
void calculateRes(_grid_p u, _grid_p f, _grid_p r){
    int i, j;
    for (i = 1; i < u->size-1; i++){
        for (j = 1; j < u->size-1; j++){
            r->grid[i][j] = f->grid[i][j] - ((8*u->grid[i][j])
                          - u->grid[i-1][j-1] - u->grid[i+1][j-1]
                          - u->grid[i-1][j+1] - u->grid[i+1][j+1]
                          - u->grid[i][j-1]   - u->grid[i-1][j]
                          - u->grid[i][j+1]   - u->grid[i+1][j] );
        }
    }
}
```

4.4 Additives Mehrgitterverfahren

Dass additive Mehrgitterverfahren nutzt im Grunde genommen die Operatoren des multiplikativen Mehrgitterverfahrens. Allerdings kommen hier die geglättete Restriktion und Prolongation ($\bar{R}, \bar{P}$) sowie der symmetrisierte Glätter ($\bar{N}$) zum Einsatz.

4.4.1 Symmetrisierter Glätter

Die Funktione `nBar` berechnet $\bar{N}$ nach Formel 3.22

```
void nBar(_grid_p u, _grid_p f){
    _grid_p help;
    help = allocateGrid(help, f->size);
    ndotb(f, help);
    gridD(help);
    ndotbT(help, u);
    freeGrid(help);
}
```

4.4.2 Geglättete Prolongation

Die geglättete Prolongation nutzt die in Kapitel 4.3.1 vorgestellte Prolongationsoperation, dabei werden die prolongierten Werte werden anschließend mit $(I - N^T A)$ geglättet. Dafür benötigt man zwei Hilfsgitter `grid` und `help`, in denen die Zwischenergebnisse gespeichert werden.

```
1  void prolongationBar(_grid_p e_c, _grid_p e_f){
2      _grid_p grid, help;
3      grid = allocateGrid(grid, e_f->size);
4      help = allocateGrid(help, e_f->size);
5      prolongation(e_c, grid);
6      gridA(grid, help);
7      ndotbT(help, e_f);
8      subGrid(grid, e_f, e_f);
9      freeGrid(grid);
10     freeGrid(help);
11 }
```

4.4.3 Geglättete Restriktion

Die geglättete Restriktion glättet die Werte des Gitters erst vor und nutzt anschließend die in Kapitel 4.3.2 vorgestellte Restriktionsoperation. Analog zur Prolongationsoperation benötigt man hierbei zwei Hilfsgitter, die anschließend nicht mehr benötigt und dementsprechend freigegeben werden.

```
1  void restrictionBar(_grid_p u_c, _grid_p r){
2      _grid_p grid, help;
3      grid = allocateGrid(grid, r->size);
4      help = allocateGrid(help, r->size);
5      ndotb(r, help);
6      gridA(help, grid);
7      subGrid(r, grid, help);
8      restriction(u_c, help);
9      freeGrid(grid);
10     freeGrid(help);
11 }
```

4.4.4 Hilfsfunktionen

Für die geglättete Prolongation und Restriktion benötigt man A

```
1  void gridA(_grid_p u, _grid_p a){
2  int i,j;
3      for (i = 1; i < u->size-1; i++){
```

```
4           for (j = 1; j < u->size-1; j++){
5               a->grid[i][j]  = ((8*u->grid[i][j])
6                                 - u->grid[i-1][j-1] - u->grid[i-1][j]
7                                 - u->grid[i-1][j+1] - u->grid[i][j-1]
8                                 - u->grid[i][j+1]   - u->grid[i+1][j-1]
9                                 - u->grid[i+1][j]   - u->grid[i+1][j+1]);
10          }
11      }
12  }
```

Die Hilfsfunktion `gridD` berechnet den Diagonalanteil von A:

```
1   void gridD(_grid_p grid){
2       int i,j;
3       for (i = 1; i < grid->size-1; i++){
4           for (j = 1; j < grid->size-1; j++){
5               grid->grid[i][j] = 8*grid->grid[i][j];
6           }
7       }
8   }
```

4.5 Die Verfahren

4.5.1 GS-Aufruf

Das iterative Gauß-Seidel-Verfahren ist einfach - man führt hintereinander mehrmals die Funktion `simpleGaussSeidel()` auf dem Arbeitsgitter u aus. Der genaue Aufruf ist in Abschnitt 4.6 beschrieben

4.5.2 MG-Aufruf/ZG-Aufruf

Der folgende Quellcode zeigt die Implementierung der Algorithmen des Zwei- (Algorithmus 3.2) und Mehrgitterverfahren (Algorithmus 3.3). Hier sieht man nochmal die enge Verwandtschaft zwischen den beiden Verfahren. In den Zeilen 5-7 wird das übergebene Gitter um `preSmooth` Iterationen mit dem Gauß-Seidel-Verfahren vorgeglättet, dementsprechend in Zeilen 21-23 um `postSmooth` Iterationen mit dem Gauß-Seidel-Verfahren nachgeglättet. Die für die nächste Gitterstufe benötigten Hilfsgitter mit halber Gitterweite werden angelegt, das Residuum wird berechnet und auf die nächstgröbere Gitterstufe restringiert. Im Falle des Zweigitterverfahrens werden 100000 Iterationen des Gauß-Seidel-Verfahrens ausgeführt. Das Mehrgitterverfahren ruft diese Funktion rekursiv auf, bis die gröbste Gitterstufe mit einem inneren Gitterpunkt erreicht wird. In dem Fall wird das Gitter durch eine Iteration des Gauß-Seidel-Verfahrens direkt gelöst. Anschließend werden die Rekursionsstufen rückwärt

durchlaufen, dabei wird das Ergebnis jeweils auf die nächstfeinere Gitterstufe prolongiert und als Korrektur verwendet. Nach Beendigung der Rekursion wurde das initiale Gitter u_f um das Ergebnis einer Iteration des Zwei- oder Mehrgitterverfahrens korrigiert. Durch wiederholten Aufruf, etwa durch das Rahmenprogramm aus Kapitel 4.6, können mehrere Iterationen des Mehrgitterverfahrens durchgeführt werden.

```
1  void multigridSolver(_grid_p u_f, _grid_p f, int mode, int preSmooth, int postSmooth){
2      _grid_p e_c, e_f, r_f, r_c;
3      int i;
4      if((u_f->size/2) > 1 ){
5          for(i = 0; i < preSmooth; i++){
6              simpleGaussSeidel(u_f, f);
7          }
8          r_f = allocateGrid(r_f, u_f->size);
9          calculateRes(u_f, f, r_f);
10         e_c = allocateGrid(e_c, (u_f->size+1)/2);
11         r_c = allocateGrid(r_c, (u_f->size+1)/2);
12         e_f = allocateGrid(e_f, u_f->size);
13         restriction(r_c, r_f);
14         if(mode == 2){
15             gridAInv(e_c, r_c, 100000);
16         } else {
17             multigridSolver(e_c, r_c, mode, preSmooth, postSmooth);
18         }
19         prolongation(e_c, e_f);
20         coarseGridCorrect(u_f, e_f);
21         for(i = 0; i < postSmooth; i++){
22             simpleGaussSeidelB(u_f, f);
23         }
24         freeGrid(e_f);
25         freeGrid(e_c);
26         freeGrid(r_c);
27         freeGrid(r_f);
28     } else {
29         simpleGaussSeidel(u_f, f);
30     }
31 }
```

4.5.3 Aufruf des additiven Mehrgitterverfahren

Das additive Mehrgitterverfahren aus Algorithmus 3.4 ist der Einfachheit halber als eine Funktion mit zwei Unterfunktionen implementiert. In Zeile 3 wird durch Aufruf der Funktion `calculateNumberofMGLevel(u)` bestimmt, wie viele Stufen die Gitterhierarchie des Gitters `u` besitzt. Damit wird in Zeile 5 das Array `error` angelegt, das zu jeder Gitterstufe einen Pointer auf das jeweilige Korrekturgitter enthält. Für die feinste Stufe wird in den Zeilen 8 und 9 der Korrekturwert für die feinste Gitterstufe berechnet. In Zeile 11 wird die Hilfsfunktion

für die Restriktion, in Zeile 12 die für die Prolongation aufgerufen. Anschließend wird das Gitter um die Summe der Korrekturgitter jeder Gitterstufe korrigiert.

```
1  void additiveMGSolver(_grid_p u, _grid_p f){
2      int i, j, k;
3      int levelCount = calculateNumberOfMGLevel(u);
4      _grid_p correction, firstLevel;
5      _grid_p * error = malloc(levelCount * sizeof(_grid_p *));
6      correction = allocateGrid(correction, u->size);
7      firstLevel = allocateGrid(firstLevel, u->size);
8      calculateRes(u, f, firstLevel);
9      nBar(correction, firstLevel);
10     freeGrid(firstLevel);
11     restrictionHelper(u, f, error, levelCount);
12     prolongationHelper(error, levelCount);
13     for(k=0; k<levelCount; k++){
14         addGrid(correction, error[k], correction);
15     }
16     coarseGridCorrect(u, correction);
17     freeGrid(correction);
18     for(i = 0; i < levelCount; i++){
19         freeGrid(error[i]);
20     }
21     free(error);
22 }
```

Die Hilfsfunktion `restrictionHelper()` stellt dabei eine rekursive Funktion dar, die, analog zum multiplikativen Mehrgitteraufruf, für jede Stufe der Gitterhierarchie das Residuum berechnet, dabei allerdings für jede Gitterstufe das mit `nbar` berechnete Korrekturgitter im Array `error`$[\ell]$ ablegt.

```
1  void restrictionHelper(_grid_p u, _grid_p f, _grid_p * error, int level){
2      if(level > 0){
3          int i, j, k;
4          _grid_p residuum, uCoarse, bCoarse, helper;
5          residuum = allocateGrid(residuum, u->size);
6          uCoarse = allocateGrid(uCoarse, ((u->size+1)/2));
7          fCoarse = allocateGrid(fCoarse, ((u->size+1)/2));
8          helper = allocateGrid(helper, ((u->size+1)/2));
9          calculateRes(u, f, residuum);
10         restrictionBar(fCoarse, residuum);
11         nBar(helper, fCoarse);
12         error[level-1] = helper;
13         restrictionHelper(uCoarse, fCoarse, error, level-1);
14         freeGrid(uCoarse);
15         freeGrid(fCoarse);
16         freeGrid(residuum);
17     }
18 }
```

Die erhaltenen Korrekturgitter liegen anschließend in der Größe der jeweiligen Gitterstufe vor. Die Hilfsfunktion `prolongationHelper()` führt die notwendigen Prolongationsschritte durch, die, entsprechend dem Algorithmus, für jede Gitterstufe einzeln vorgenommen werden müssen. Dabei werden die Pointer auf das Korrekturgitter einfach durch einen neuen Pointer ersetzt, der auf das prolongierte Korrekturgitter zeigt.

```
1  void prolongationHelper(_grid_p * error, int level){
2      int i,j,k,l;
3      _grid_p helper;
4      for(k = 0; k < level; k++){
5          for(l = 0; l <= k; l++){
6              helper = allocateGrid(helper, ((error[l]->size*2)-1));
7              prolongationBar(error[l], helper);
8              freeGrid(error[l]);
9              error[l] = helper;
10         }
11     }
12 }
```

Diese Variante hat den Vorteil, dass sie sehr einfach ist. Da aber jede Stufe ein eigenes Korrekturgitter hat, benötigt diese Hilfsfunktion sehr viel Speicherplatz (siehe Kapitel 5.2).

4.6 Das Rahmenprogramm

Damit die beschriebenen Verfahren als iterative Verfahren verwendet werden können, ist ein wiederholter Aufruf derselben notwendig. Dieser wird durch eine Schleife innerhalb des Hauptprogramms realisiert. Der folgende Abschnitt zeigt den Ausschnitt aus der Hauptfunktion, die die iterativen Aufrufe ermöglicht. Der Code zum Initialisieren der Speicherstruktur, der Parameter und der Messungen wurde entfernt, um den Codeausschnitt übersichtlicher zu machen.

```
1
2  int main( int argc, char* argv[] ) {
3      ...
4      for(counter = 0; counter < iterations; counter++){
5          switch(mode) {
6              case 1: simpleGaussSeidel(u,f);
7                      break;
8              case 2: multigridSolver(u, f, true, mode, preSmooth, postSmooth);
9                      break;
10             case 3: multigridSolver(u, f, true, mode, preSmooth, postSmooth);
11                     break;
12             case 4: additiveMGSolver(u,f);
13                     break;
14             default: printf("-- Diesen Modus gibt es nicht --\n");
15                     break;
```

```
16          }
17      }
18      ...
19      printf("Programm beendet.\n");
20      return 0;
21  }
```

Alle im Rahmen dieser Arbeit implementierten Funktionen stehen über das Rahmenprogramm zur Verfügung, die Steuerung erfolgt dabei über folgende Parameter:

```
./main gridWidth iterations sides mode preSmooth postSmooth path
```

- `gridWidth`: Weite des (feinsten) Gitters
- `iterations`: Anzahl der Iterationen
- `sides`: Schalter für die Randwerte
- `mode`: Schalter für den zu verwendenden Löser
- `pre-/postSmooth`: Anzahl der Vor- und Nachglättungsschritte
- `path`: Pfad zum Ablageverzeichnis für Ergebnisdateien

Da für alle Verfahren ein und dasselbe Programm verwendet werden soll, wird der Parameter `mode` als Schalter für das jeweilige Verfahren verwendet. 1 bedeutet dabei, dass das Gauss-Seidel-Verfahren zum Einsatz kommt, 2 ist das Zweigitterverfahren, 3 das multiplikative und 4 das additive Mehrgitterverfahren. Der Schalter `sides` bietet die Optionen 1 für $u|_\Gamma = 0$ und 1 für $u|_\Gamma = (\sin(\frac{j \cdot \Pi}{u_{dim}}) \cdot \exp(\frac{i \cdot \Pi}{u_{dim}}))$.

Zur einfacheren Verwendung lässt sich das Programm alternativ über den Aufruf

```
./main manual
```

interaktiv ausführen, wobei alle Parameter abgefragt werden.

5 Ergebnisse

5.1 Fehler- und Konvergenzanalyse

Die gesamten Ergebnisse wurden auf einem `Intel Core i5 2520M` mit `16GB 1333MHz DDR3 RAM` unter `Linux Mint` mit Kernelversion `3.16.0-38` generiert. Dabei wurden der Quellcode mit dem Compiler `GGC 4.8.4` und den Flags `-lm -Wall` (siehe auch Makefile) kompiliert.

5.1.1 Diskretisierungsfehler

Der Diskretisierungsfehler d der $m$ten Iteration wird durch die Formel

$$d^m = \sqrt{\frac{1}{u_{dim}^2} \cdot \sum_{i,j=1}^{u_{dim}-1} (u_{i,j}^m - c_{i,j})^2} \tag{5.1}$$

mit

- u_{dim}, die Gitterweite $n = 1/h$ und
- c, dem Gitter, das die erwartete Lösung enthält

definiert. Die Berechnung dieser Werte ist besonders für die Überprüfung der Korrektheit der Verfahren von Interesse. Unabhängig von Konvergenz- und Ausführungsgeschwindigkeit sollten alle approximativen Löser letztendlich gegen den gleichen Lösungswert konvergieren. Um nicht jeden Gitterpunkt einzeln vergleichen zu müssen, nutzt man die Norm über das gesamte Gitter, wie in Formel 5.1 dargestellt. Ein Messwert, der schneller zu berechnen ist und damit weniger Einfluss auf die Laufzeit hat, ist der Diskretisierungsfehler des Gitterpunktes $(1/2, 1/2)$:

$$d_{1/2,1/2}^m = u_{1/2,1/2}^m - c_{1/2,1/2} \tag{5.2}$$

Die folgenden Tabellen wurden mit folgendem Aufruf des Programms erzeugt:

```
./main u_dim 1000 2 mode 1 1 ../data/
```

Die Verfahren werden damit 1000 Iterationen ausführen, wobei im Falle des Zwei- und Mehrgitterverfahrens jeweils ein Vor- und Nachglättungsschritt stattfindet. Die Ergebnisse werden nach `../data/` geschrieben. Das Programm approximiert dabei das lineare Gleichungssystem $Au = f$ mit $f = 0$ und $u_{|\Gamma} = (\sin(\frac{j \cdot \Pi}{u_{dim}}) \cdot \exp(\frac{i \cdot \Pi}{u_{dim}}))$. Die Werte der erwarteten Lösung c berechnen sich dementsprechend wie die Randwerte von u.

Tabelle 5.1: Diskretisierungsfehler der Verfahren

u_{dim}	d_{GS}	d_{ZG}	d_{MG}	d_{BPX}
5	0.173228	0.173228	0.173228	0.173228
9	0.046718	0.046718	0.046718	0.046718
17	0.012251	0.012251	0.012251	0.012251
33	0.003148	0.003148	0.003148	0.003148
65	0.000798	0.000798	0.000798	0.000798
129	0.000201	0.000201	0.000201	0.000201
257	0.000050	0.000050	0.000050	0.000050
513	0.000013	0.000013	0.000013	0.000013
1025	0.000003	0.000003	0.000003	0.000003

Die Werte in Tabelle 5.1 zeigen, dass alle Verfahren erwartungsgemäß gegen gleichen Wert, in etwa $O(h^2)$ konvergieren. Damit liefern die implementierten Funktionen korrekte Ergebnisse und sind demnach korrekt implementiert.

5.1.2 Konvergenzraten

Um festzustellen, wie schnell ein Verfahren konvergiert, bestimmt man λ zu

$$\lambda^m = \sqrt{\frac{1}{u_{dim}^2} \cdot \sum_{i,j=1}^{u_{dim}-1} (u_{i,j}^m)^2} \tag{5.3}$$

und daraus die Konvergenzrate

$$\lambda = \lim_{m \to \infty} \frac{\lambda^{m+1}}{\lambda^m} \tag{5.4}$$

Da $\lambda \to 1$ für $h \to 0$, betrachtet man zusätzlich ρ:

$$\rho^m = 1 - \frac{\lambda^{m+1}}{\lambda^m} \tag{5.5}$$

Um zu verhindern, dass man ungenaue Ergebnisse aufgrund der eingeschränkten Maschinengenauigkeit erhält, kann man nach jedem Iterationsschritt das Arbeitsgitter u_m mit $1/\lambda$ skalieren, sodass

$$\rho^m = 1 - \lambda^m \tag{5.6}$$

Die folgenden Tabellen wurden mit folgendem Aufruf des Programms erzeugt:

```
./main u_dim 1000 1 mode 1 1 ../data/
```

Dadurch werden alle Verfahren 1000 mal wiederholt, wobei im Falle des Zwei- und Mehrgitterverfahrens jeweils ein Vor- und Nachglättungsschritt stattfindet. Die Ergebnisse werden nach `../data/` geschrieben. Dabei approximiert das Programm das lineare Gleichungssystem $Au = f$ mit $f = 0$ und $u_{|\Gamma} = 0$. Die erwartete Lösung c ist dementsprechend ein Nullgitter mit $u_{i,j} = 0$. In jedem Iterationsschritt berechnet man mit den Formeln 5.3 und 5.6 die Werte der Konvergenzrate λ.

Für das Gauß-Seidel-Verfahren sind keine besonders guten Konvergenzraten zu erwarten. Diese sollten bei den Mehrgitterverfahren allerdings deutlich besser sein.

Tabelle 5.2: Konvergenzraten der Verfahren

u_{dim}	ρ_{GS}	ρ_{ZG}	ρ_{MG}	ρ_{BPX}
5	0.620803	0.900755	0.900755	0.900755
9	0.208364	0.855857	0.847202	0.847202
17	0.056322	0.838892	0.832469	0.832469
33	0.014362	0.834304	0.828810	0.828810
65	0.003608	0.839576	0.828725	0.828725
129	0.000903	0.836463	0.828463	0.828463
257	0.000226	0.835878	0.828333	0.828333
513	0.000057	0.836320	0.828339	0.828339
1025	0.000018	0.836399	0.828279	0.828279

Es zeigt sich dass, wie erwartet, das Gauß-Seidel-Verfahren am langsamsten konvergiert, während die Konvergenzrate des additiven und des multiplikativen Mehrgitterverfahrens identisch und dabei von der Feinheit der Gitter, also h, unabhängig ist. Daher funktioniert das additive Mehrgitterverfahren genauso zuverlässig wie das multiplikative Mehrgitterverfahren, bietet dabei aber aufgrund seiner einfachen Parallelisierbarkeit einen potentiellen Vorteil. Die Konvergenzraten des Zweigitterverfahrens sind sehr gut, dabei benötigt das Verfahren aufgrund der hohen Anzahl der Gauß-Seidel-Iterationen auf der Grobgitterstufe jedoch sehr viel Rechenzeit.

Auf den ersten Blick scheint das additive Mehrgitterverfahren also ein ideales Verfahren zu sein: Es hat gute besten Konvergenzeigenschaften und ist dabei einfach parallelisierbar. Bei der genaueren Betrachtung fallen jedoch Nachteile dieser Implementierung ins Auge. Diese werden im Folgenden vorgestellt.

5.2 Laufzeit- und Speicheranalyse

Die vorgestellten Verfahren unterscheiden sich teilweise deutlich in Speicher- und Rechenzeitbedarf. Die folgenden Abschätzungen geben dabei den maximal zu erwartenden Speicherverbrauch beziehungsweise die erwartete Laufzeit wieder.

5.2.1 Speicherbedarf

Der Speicherbedarf eines $n \times n$-Gitters lässt sich durch

$$\mathtt{M}_{u^n} = n^2 \cdot \text{sizeof } \texttt{double} \tag{5.7}$$

berechnen. Damit ergibt sich für ein Gitter mit 1025×1025 Gitterpunkte ein theoretischer Speicherbedarf von etwa 8 Megabyte.

Das Gauß-Seidel-Verfahren arbeitet *in-place*, dementsprechend erwartet man einen sehr niedrigen Speicherbedarf, der sich eigentlich nur aus den beiden notwendigen Gittern u und f zusammen. Daraus ergibt sich für das Gauß-Seidel-Verfahren ein Speicherbedarf von

$$\mathtt{M}_{\text{GS}} = 2 \cdot \mathtt{M}_\ell, \tag{5.8}$$

Dabei steht $\mathtt{M}_\ell$ für den Speicherbedarf eines $n \times n$-Gitters, dementsprechend steht $\mathtt{M}_{\ell-1}$ für den Speicherbedarf eines $\frac{n}{2} \times \frac{n}{2}$-Gitters, wobei dieses aufgrund der halbierten Gitterweite nur ein Viertel des Speicherbedarfs besitzt. Der Speicherbedarf des kleinstmöglichen Gitters, eines 3×3-Gitters, wird mit $\mathtt{M}_0$ bezeichnet.

Das Zwei- und Mehrgitterverfahren benötigt schon mehr Speicher, beim Zweigitterverfahren benötigt man die beiden Gitter der nächstgröberen Stufe, beim Mehrgitterverfahren die der gesamten Hierarchie von Gleichungssystemen. Zudem benötigt man pro Stufe jeweils ein Gitter für das Residuum und die Korrekturwerte. Dies bedeutet für das Zweigitterverfahren einen Speicherbedarf von

$$\mathtt{M}_{\text{ZG}} = 4 \cdot \mathtt{M}_\ell + 2 \cdot \mathtt{M}_{\ell-1} = 4.5 \cdot \mathtt{M}_\ell \tag{5.9}$$

Bei einem u^0 gleicher Gitterweite benötigt das Zweigitterverfahren also 2.25 mal mehr Speicher als das Gauß-Seidel-Verfahren.

Für das multiplikative Mehrgitterverfahren berechnet sich der Speicherbedarf aufgrund der Abschätzungen aus Kapitel 3.2 und der pro Gitterstufe benötigten Hilfsgitter zu

$$\mathtt{M}_{\text{mMG}} = 4 \cdot \mathtt{M}_{\ell} + 4 \cdot \mathtt{M}_{\ell-1} + 4 \cdot \mathtt{M}_{\ell-2} + \cdots + 2 \cdot \mathtt{M}_{0} \approx 5.5 \cdot \mathtt{M}_{\ell} \tag{5.10}$$

Für das additive Mehrgitterverfahren teilt sich der Speicherbedarf in den Speicherbedarf der im Rahmen der Implementierung (Kapitel 4.5.3) gezeigten Hilfsfunktionen auf.

Für die Hilfsfunktion, die die Restriktion über alle Gitterstufen übernimmt, liegt der Speicherbedarf bei

$$\mathtt{M}_{\text{RHelper}} = 1 \cdot \mathtt{M}_{\ell} + 4 \cdot \mathtt{M}_{\ell-1} + 4 \cdot \mathtt{M}_{\ell-2} + \cdots + 3 \cdot \mathtt{M}_{0} \approx 2.5 \cdot \mathtt{M}_{\ell} \tag{5.11}$$

Dabei wird auf jeder Gitterstufe ℓ ein Gitter der Größe $\mathtt{M}_{\ell}$ an das Korrekturarray übergeben. Damit hat das Korrekturarray zu diesem Zeitpunkt die Größe

$$\mathtt{M}_{\text{CArray}} = \mathtt{M}_{\ell-1} + \mathtt{M}_{\ell-2} + \cdots + \mathtt{M}_{0} \approx 1.5 \cdot \mathtt{M}_{\ell} \tag{5.12}$$

Für die Hilfsfunktion, die die Prolongation über alle Gitterstufen übernimmt, wird im Grunde genommen kein Speicher benötigt, da diese nur den Inhalt des Korrekturarrays schrittweise prolongiert, bis alle Gitter die gleiche Größe wie u_0 haben. Allerdings erhöht sich durch die Ausführung der Speicherbedarf des Korrekturarrays, dadurch ergibt sich ein Speicherbedarf von

$$\mathtt{M}_{\text{CArray}} = \ell - 1 \cdot \mathtt{M}_{\ell} \tag{5.13}$$

Der Speicherbedarf von $\ell - 1 \cdot \mathtt{M}_{\ell}$ ergibt sich daraus, dass das Korrekturgitter der feinsten Gitterebene bereits vorher in der Hauptfunktion berechnet wird. Daher ergibt sich für die Hauptfunktion einen Speicherbedarf von

$$\mathtt{M}_{\text{aMG}} \approx 3 \cdot \mathtt{M}_{\ell} + \mathtt{M}_{\texttt{max,Hilfsfunktion}} \tag{5.14}$$

Da die Hilfsfunktionen sequentiell ablaufen, wird der Speicher nicht zur gleichen Zeit belegt. Daher ist nur die Hilfsfunktion, die den größten Speicherbedarf hat, für die Berechnung des maximalen Speicherbedarfs interessant. Am meisten Speicher wird durch die Hilfsfunktion benötigt, die die Werte des Korrekturarrays prolongiert.

Damit ergibt sich ein Speicherbedarf von

$$\mathtt{M}_{\text{aMG}} \approx 3 \cdot \mathtt{M}_{\ell} + \mathtt{M}_{\texttt{CArray}} = ((\ell - 1) + 3) \cdot \mathtt{M}_{\ell} \tag{5.15}$$

Dadurch fällt der Speicherbedarf dieser Implementierung des additiven Mehrgitterverfahrens deutlich größer aus als der der anderen Verfahren. Es zeigt sich also, dass das Gauß-Seidel-, das Zweigitter- sowie das multiplikative Mehrgitterverfahren bezüglich des Speicherbedarfs alle

in einer Größenordnung liegen, diese Implementierung des additiven Mehrgitterverfahrens jedoch aufgrund des Korrekturarrays deutlich mehr Speicher benötigt.

Wenn man also das Verfahren effizienter gestalten möchte, so muss man ein intelligenteres Korrekturgitter-Management finden. Eine Möglichkeit wäre, die Korrekturgitter, sobald sie zur Verfügung stehen, zu prolongieren und auf das Ausgangsgitter zu addieren. Das könnte, je nach Implementierungsqualität, den Speicherbedarf etwas senken. Nichtsdestotrotz braucht man die Korrekturgitter weiterhin getrennt für jede Stufe der Gleichungssystem-Hierarchie.

Speicherbedarfsmessungen haben das Problem, dass sie sehr ungenau sind. Während man bei Laufzeitmessungen die Anzahl an CPU-Zyklen zählen kann, lässt sich der Speicherbedarf nur grob anhand der Speicherauslastung abschätzen. Die Werte in Tabelle 5.3 wurden bei deaktivierter `swap`-Datei mit `top` gemessen.

Tabelle 5.3: Speicherbedarf der Verfahren

	u_{dim}	M_u	M_{GS}	M_{ZG}	$\mathsf{M}_{\mathsf{mMG}}$	$\mathsf{M}_{\mathsf{aMG}}$
Berechnet	1025	8 MB	16 MB	36 MB	44 MB	96 MB
Gemessen	1025	≈9 MB	≈20 MB	≈40 MB	≈50 MB	≈110 MB
Theoretisch	8193	512 MB	1024 MB	2304 MB	2816 MB	7168 MB
Berechnet	8193	≈550 MB	≈1200 MB	≈2500 MB	≈3000 MB	≈7600 MB

Der Grund für die teilweise deutliche Abweichung von der Schätzung dürfte neben der ungenauen Messung auch die Allokation von Arrays für die Historie der Analysevariablen sein, da pro Programmaufruf mehrere Felder dafür alloziert werden. Trotzdem stellen die Schätzungen einen guten Näherungswert für den maximalen Speicherbedarf der Verfahren dar.

Für eine parallele Implementierung könnte die Speicherproblematik, je nach System, mehr oder weniger wichtig werden. Jede Stufe der Gitterhierarchie kann, abgesehen von der Restriktion, unabhängig von den anderen Stufen berechnet werden und erzeugt dabei ein eigenes Korrekturgitter. Für einen Rechencluster, bei dem jeder Knoten eigenen Speicher besitzt, stellt dies möglicherweise kein Problem dar. Bei einem Rechner mit gemeinsam genutztem Speicher könnte jedoch die gleiche Problematik wie bei der vorliegenden sequentiellen Implementierung auftreten: Die Korrekturgitter liegen parallel im Speicher.

5.2.2 Laufzeit

Die im Rahmen dieser Arbeit vorgenommene Implementierung wurde in erster Linie auf geringen Speicherbedarf hin optimiert. Dabei wurde in Kauf genommen, dass Speicher häufig

alloziert und freigegeben wird. Dies kann einen großen Einfluss auf die Laufzeit des Lösers haben.

Um die Laufzeiteigenschaften abschätzen zu können, wurden die Laufzeiten von drei wichtigen Operationen, der Allokation, einer Gauß-Seidel-Iteration sowie der Füllung mit einem Wert, für verschieden große Gitter gemessen.

Tabelle 5.4: Dauer der Operationen in Sekunden

u_{dim}	$t_{\text{alloc}}[s]$	$t_{\text{GS}}[s]$	$t_{\text{fill}}[s]$
17	0.000009	0.000010	0.000004
33	0.000040	0.000040	0.000008
65	0.000081	0.000100	0.000029
129	0.000286	0.000700	0.000116
257	0.001522	0.002900	0.000585
513	0.006675	0.011790	0.002597
1025	0.021705	0.040000	0.007739
2049	0.062311	0.180000	0.023989
4097	0.152018	0.750000	0.072330
8193	0.444205	1.101950	0.288820
16385	1.779012	4.284990	1.161819

In Tabelle 5.4 sieht man die gemessene Zeit in Sekunden für diese Operationen. Auffällig dabei ist, dass die Allokation eines Speicherbereichs in der gleichen Größenordnung wie eine Operation über das gesamte Gitter abgeschlossen wird - für ein Gitter mit 513×513 Gitterpunkten benötigt die Allokationsoperation $\mathsf{T}_{\text{alloc},513} = 6.675\text{ms}$, eine Iteration des einfachen Gauß-Seidel-Verfahrens benötigt demgegenüber $\mathsf{T}_{\text{GS},513} = 11.79\text{ms}$. Dabei zeigt sich, dass für kleine Gitter die Allokation länger dauert als eine Iteration des einfachen Gauß-Seidel-Verfahrens, für größere Gitter dreht sich dieses jedoch ins Gegenteil. Mit steigender Gittergröße fällt die Allokationszeit eines Gitters deutlich unter $1/2 \cdot \mathsf{T}_{\text{GS}}$. Daher macht sich das häufige Allozieren und Freigeben des Speichers zwar bemerkbar, mit steigender Gittergröße wird der Einfluss davon jedoch schwächer. Da die Implementierung zu dieser Arbeit auf geringen Speicherbedarf hin optimiert wurde, wurde der größere Zeitbedarf als notwendiges Übel akzeptiert.

Da alle in dieser Arbeit vorgestellten Löser iterativ arbeiten, für ein gutes Ergebnis also mehrfach angewendet werden müssen, ist zur Laufzeitabschätzung interessant, wie lange eine Iteration des Lösers braucht.

u_{dim}	GS ●	ZG ■	mMG ●	aMG ∗
17	0.00001	0.07000	0.00003	0.00012
33	0.00004	0.33000	0.00014	0.00051
65	0.00010	1.00000	0.00050	0.00200
129	0.00070	5.0000	0.00230	0.01100
257	0.00290	24.0000	0.00790	0.05200
513	0.01179	96.0000	0.04170	0.24100
1025	0.04000	400.000	0.15000	1.00000
2049	0.18000	1613.00	0.62000	4.50000
4097	0.75000	7500.00	2.58000	19.80000

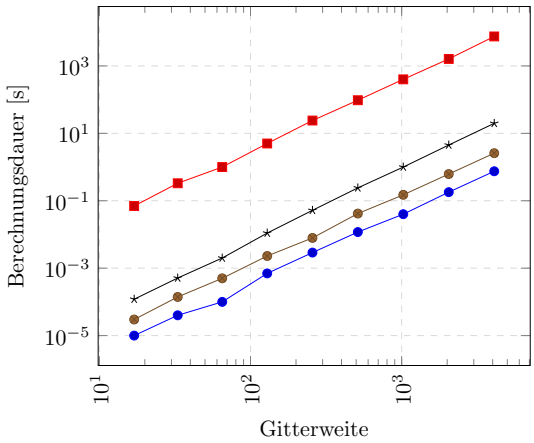

Abbildung 5.1: Berechnungsdauer für eine Iteration der Verfahren in Sekunden: Gauß-Seidel-Verfahren●, Zweigitterverfahren ■, multiplikatives Mehrgitterverfahren ● und additives Mehrgitterverfahren∗

Die Tabelle in Abbildung 5.1 listet die Zeit auf, die die Berechnung von jeweils einer Iteration der verschiedenen Verfahren für unterschiedliche Gitterdimensionen benötigt. Dabei ist die Berechnung von einer Iteration des Gauss-Seidel-Verfahrens grundsätzlich am schnellsten, sie korrigiert jedoch auch am wenigsten. Etwa die vierfache Zeit einer Gauß-Seidel-Iteration benötigt eine Iteration des multiplikativen Mehrgitterverfahrens. Eine Iteration des additiven Mehrgitterverfahrens benötigt wiederum davon die vierfache Zeit, also in etwa die 16-fache Zeit einer Iteration des Gauß-Seidel-Verfahrens. Das Zweigitterverfahren fällt aufgrund der großen Zahl an Gauß-Seidel-Iterationen auf der Grobgitterstufe weit zurück. Damit kann man folgende Abschätzung aufstellen:

$$\mathsf{T}_{\text{GS}} \approx {}^1\!/_4 \cdot \mathsf{T}_{\text{mMG}} \approx {}^1\!/_{16} \cdot \mathsf{T}_{\text{aMG}} \tag{5.16}$$

$$\mathsf{T}_{\text{aMG,seq}} = \mathsf{T}_\ell + \mathsf{T}_{\ell-1} + \cdots + \mathsf{T}_1 + \mathsf{T}_0 \tag{5.17}$$

$$\mathsf{T}_{\text{aMG,par}} = \mathsf{T}_\ell + \mathsf{T}_{\text{Grobgitterkorrektur}} + \mathsf{T}_{\text{Restriktion}} \tag{5.18}$$

Die Laufzeit des additiven Mehrgitterverfahrens könnte möglicherweise durch parallele Implementierung stark reduziert werden. Bei idealer Implementierung mit ℓ parallel nutzbaren Rechenkernen könnte man die durch die sequentielle Berechnung jeder Gitterstufe notwendige Rechenzeit $\mathsf{T}_{\text{aMG,seq}}$ auf $\mathsf{T}_{aMG,par}$ reduzieren. Dabei kann man wie in Kapitel 3.3 beschrieben die Restriktion nicht parallel ausführen, trotzdem sollte man die Rechenzeit des multiplikativen Verfahrens unterbieten können.

6 Zusammenfassung und Ausblick

Die Lösungsapproximation ist durch die implementierten Verfahren, das Gauß-Seidel-Verfahren, das multiplikativen und das additiven Mehrgitterverfahren möglich. Dass das einfache Gauß-Seidel-Verfahren in Bezug auf Konvergenzgeschwindigkeit nicht mit den Mehrgitterverfahren mithalten kann, stand von vornherein fest.

Die Annahme, dass das additive Mehrgitterverfahren in der vorgestellten Form exakt die gleichen Ergebnisse liefert wie das multiplikative Mehrgitterverfahren, wurde bestätigt. Dabei wurde festgestellt, dass, abgesehen von der Speicherproblematik dieser Implementierung, das additive Mehrgitterverfahren das Potential besitzt, die Lösungsapproximation deutlich zu beschleunigen, ohne die Stabilität der Lösung zu gefährden.

Die vorliegende Implementierung bietet mit ihrem einfachen Speichermanagement und den vorhandenen Funktionen eine gute Basis, die man durch eine Parallelisierung des additiven Mehrgitterverfahrens weiter optimieren kann. Die erwartete Beschleunigung der Lösungsapproximation ist somit durch eine parallele Implementierung der additive Mehrgitterverfahren realisierbar.

Ausblick

Das additive Mehrgitterverfahren ermöglicht eine neue Geschwindigkeitsstufe für approximative Löser. Da man jedoch anhand der Implementierung und der daraus resultierenden Ergebnisse davon ausgehen muss, dass eine Lösung der beschriebenen Probleme einen erheblichen Mehraufwand in der Speicherverwaltung bedeuten würde, der sich wiederum in der Laufzeit niederschlagen würde, ist eine Optimierung der Speicherzugriffe notwendig, bevor diese Implementierung des additiven Mehrgitterverfahrens ihre Stärken gegenüber dem multiplikativen Mehrgitterverfahren ausspielen kann.

Literaturverzeichnis

[Hac91] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Teubner, 1991. (Zitiert auf den Seiten 11 und 13)

[Meh15] M. Mehl. *Vorlesungsskript Grundlagen des Wissenschaftlichen Rechnens*. 2015. (Zitiert auf Seite 10)

[VY14] P. S. Vassilevski, U. M. Yang. Reducing communication in algebraic multigrid using additive variants. *Numer. Linear Algebra Appl*, 21: 275–296. doi: 10.1002/nla.1928, 2014. (Zitiert auf den Seiten 13, 18 und 19)

Erklärung

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

Ort, Datum, Unterschrift