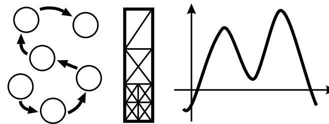
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)



Schnelle Berechnung bioelektrischer Felder

Matthias Hofmann

Studienarbeit

Schnelle Berechnung bioelektrischer Felder

Matthias Hofmann
Studienarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Marcus Mohr
Bearbeitungszeitraum:	Juni - November 2002

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 4. Dezember 2002 ....

Inhaltsverzeichnis

Einführung **5**

1 Das CG-Verfahren **6**
1.1 CG ohne Vorkonditionierung . . . 6
1.2 CG mit Vorkonditionierung . . . 8

2 Die *hypre*-Programmbibliothek **9**

3 Die *hypre*-Vorkonditionierer *ParaSails* und *Euclid* **11**
3.1 ParaSails . . . 11
3.2 Euclid . . . 13

4 Operation Count **15**
4.1 Operation Count ohne Vorkonditionierung . . . 15
4.2 Operation Count mit Vorkonditionierung . . . 16
4.2.1 Vorkonditionierung mit ParaSails . . . 16
4.2.2 Vorkonditionierung mit Euclid . . . 17

5 Das 2D-Poisson-Problem **17**
5.1 Problembeschreibung . . . 17
5.2 Testergebnisse . . . 19

6 Das verallgemeinerte 2D-Poisson-Problem **25**
6.1 Problembeschreibung . . . 25
6.2 Testergebnisse . . . 27

7 Das 2D-Slice-Problem **28**
7.1 Problembeschreibung . . . 28
7.2 Testergebnisse . . . 29

8 Das 3D-Kopfproblem **32**
8.1 Problembeschreibung . . . 32
8.2 Parameterstudie . . . 33
8.3 Testergebnisse . . . 36

9 Das Varga-Problem **40**
9.1 Problembeschreibung . . . 40
9.2 Testergebnisse . . . 42

Zusammenfassende Bemerkungen **46**

Literatur **48**

Einführung

Das Lösen von großen linearen Gleichungssystemen (LGS) der Form $Ax = b$, wobei $A \in \mathbb{R}^{n,n}$ eine Matrix, x ein unbekannter Vektor und b ein gegebener Vektor ist, stellt eine wichtige Aufgabe im naturwissenschaftlichen Bereich und im Ingenieurwesen dar. Oft entstehen solche Gleichungssysteme durch die Diskretisierung von partiellen Differenzialgleichungen, wie sie u. a. in der Strömungsmechanik auftreten. Ziel ist es, auf diese Weise eine numerische Lösung der Differenzialgleichung zu finden.

Durch einen solchen Diskretisierungsvorgang, der Voraussetzung für eine Behandlung des Problems am Computer ist, ergeben sich zumeist sehr große, jedoch nur dünn-besetzte, d. h. wenige von Null verschiedene Einträge enthaltende, Koeffizientenmatrizen. Bei dieser besonderen Art von LGS sind direkte Lösungsverfahren, wie beispielsweise die Gauß-Elimination, eher ungeeignet, da hier die "Dünnbesetztheit" der Matrix zerstört wird und dadurch ein viel zu großer Berechnungsaufwand entsteht.

Weitaus effizienter arbeiten in diesem Fall iterative Methoden, wie z. B. das Verfahren der Konjugierten Gradienten (Conjugate Gradients, CG), die durch Anwendung eines bestimmten Algorithmus, der auf der Berechnung von sich der tatsächlichen Lösung immer mehr annähernden Approximationen beruht und die "Dünnbesetztheit" von A berücksichtigt, mit vergleichsweise geringem Aufwand sehr gute Ergebnisse liefern können.

Das für diese Arbeit wichtige CG-Verfahren wird jedoch selten ohne Vorkonditionierung angewandt. Vorkonditionierer sind dabei Algorithmen, die gewisse Eigenschaften des zu lösenden LGS, bzw. der Koeffizientenmatrix A, verbessern, um eine schnellere Konvergenz, und damit möglichst eine geringere Rechenzeit zu erreichen. Die Analyse, das Testen und die Bewertung zweier bestimmter Vorkonditionierer, die beide Bestandteil der am Lawrence Livermore National Laboratory in Kalifornien entwickelten *hypre*-Programmbibliothek sind, ist das grundlegende Thema dieser Studienarbeit. Dabei baut das eine Verfahren – *Euclid* – auf einer inkompletten LU-Zerlegung der Koeffizientenmatrix auf, während das andere – *ParaSails* – eine "sparse approximate inverse"-Methode verwendet.

Die zentrale Problemstellung und Motivation der Tests kommt aus dem Bereich der Medizin: die modellbasierte Rekonstruktion der elektrischen Aktivität innerhalb des menschlichen Gehirns anhand von an einer gewissen Anzahl von Elektroden gewonnenen EEG-Messdaten – eine Aufgabe, die auf dem Gebiet der Neurologie und Neurochirurgie für Diagnose und Operationsplanung, beispielsweise bei Epilepsie- oder Tumorpatienten, von wachsender Bedeutung ist. Die möglichst effiziente Berechnung solcher Spannungsfelder ist ein aktives Forschungsgebiet, wobei die komplexe und individuelle Geometrie des menschlichen Kopfes und die unterschiedlichen Leitfähigkeiten der verschiedenen Gewebearten die größte Herausforderung darstellen.

Da sich dieser Problemkomplex von anderen Gebieten der Simulation insofern unterscheidet, dass die Rekonstruktion der Hirnaktivität (Dipollokalisation) kein klassisches Randwertproblem mit vorgeschriebenen Quellen und Randbedingungen ist, sondern die Rekonstruktion der Quellen und/oder der Randbedingungen die zentrale Aufgabe darstellt, spricht man hier von einem *inversen* Problem [16]. Verfahren, um dieses inverse Problem zu lösen, schließen immer die Lösung von sog. Vorwärtsproblemen ein. Ein einzelnes Vorwärtsproblem besteht aus der Berechnung der Potenzialverteilung, die sich im Kopf ergibt, wenn bei zwei ausgewählten Elektroden die eine als Stromquelle und die andere als Senke modelliert wird. Dies bedeutet das Lösen eines 3D-Poisson-Problems, das diskretisiert wird, indem der Kopf in kleine Würfel – sog. Voxel – eingeteilt wird [16]. Das daraus resultierende große dünnbesetzte LGS ist für iterative Lösungsverfahren mit Vorkonditionierung geeignet. Da, wie bereits erwähnt, die Lösung des inversen Problems stets die Lösung einer beträchtlichen Zahl an Vorwärtsproblemen mit jeweils unterschiedlichen rechten Seiten b erfordert, sind effiziente Methoden für diese Unteraufgabe von großer Bedeutung.

Neben dem Testen und Benchmarking von *Euclid* und *ParaSails* als Vorkonditionierer für das CG-Verfahren beim Lösen von insgesamt 26 solcher Vorwärtsprobleme anhand des Kopfmodells eines echten Patienten, wobei verschiedene Rechnerarchitekturen betrachtet werden (Abschnitt8), sollen in dieser Arbeit das Verhalten und die Eigenschaften beider Vorkonditionierer auch noch bei einigen anderen ähnlichen Problemen, die größtenteils auf das zentrale Thema hinführen, untersucht werden. Ausgehend vom einfachen 2D-Poisson-Problem (Abschnitt 5) wird die Aufgaben-

stellung schrittweise erweitert und den realistischen Gegebenheiten angepasst, um über das verallgemeinerte 2D-Poisson-Problem (Abschnitt 6) und das Slice-Problem (Abschnitt 7) – dort dienen die Gewebedaten und die Form eines echten Kopfquerschnitts als Grundlage – schließlich beim dreidimensionalen Kopfproblem anzukommen. Zum Abschluss wird in Abschnitt 9 noch das von Varga in [18] vorgestellte Problem aus der Reaktorphysik untersucht – eine ähnliche Problemstellung, die jedoch auf einer Helmholtz- anstelle einer Poisson-Gleichung wie zuvor basiert. Die ersten vier Abschnitte der Studienarbeit dienen der Beschreibung und Erläuterung einiger wichtiger theoretischer Grundlagen, Voraussetzungen und Hintergründe. Hierbei stehen das CG-Verfahren und die verwendeten Vorkonditionierer – vor allem auch im Kontext der *hypre*-Programmbibliothek – im Vordergrund.

1 Das CG-Verfahren

Da das CG-Verfahren in unseren Tests das Grundlösungsverfahren ist, auf das dann die Vorkonditionierer angewandt werden, wird im folgenden Abschnitt etwas näher darauf eingegangen.

1.1 CG ohne Vorkonditionierung

Das CG-Verfahren gehört zur Klasse der nicht-stationären iterativen Methoden zur Lösung von linearen Gleichungssystemen. Daneben gibt es noch die stationären iterativen Methoden, wozu beispielsweise das Jakobi-, das Gauß-Seidel oder das SOR (successive overrelaxation)-Verfahren gehören. Beiden Klassen gemeinsam ist der prinzipielle Lösungsvorgang: es werden sukzessive, in gewisser Weise voneinander abhängige Approximationen berechnet, um in jedem Iterationsschritt der genauen tatsächlichen Lösung immer näher zu kommen. Abgesehen davon, dass stationäre Verfahren älter und zumeist leichter zu verstehen und zu implementieren sind, besteht der Hauptunterschied darin, dass bei den Berechnungen der nicht-stationären Methoden zusätzliche Informationen verwendet werden, die sich in jedem Iterationsschritt ändern [1]. Dies bedeutet, dass bestimmte Hilfskomponenten wie Konstanten oder spezielle Vektoren jedesmal aktualisiert werden müssen.

Das CG-Verfahren, das in gewisser Hinsicht eine Erweiterung und Verbesserung der Steepest-Descent-Methode ("Steilster Abstieg") darstellt, ist eines der bekanntesten Verfahren, wenn es um die Lösung von dünnbesetzten LGS der Form $Ax = b$ (A Koeffizientenmatrix, x Vektor der Unbekannten, b Vektor der rechten Seite) geht. Eigentlich wurde es so konzipiert, dass es nur dann erfolgreich ist, wenn die Koeffizientenmatrix A drei wichtige Vorraussetzungen erfüllt: sie sollte quadratisch, symmetrisch und positiv-definit[1] sein. Allerdings reicht beispielsweise auch eine symmetrische positiv-semidefinite Matrix aus (wie wir in unseren Tests noch sehen werden).

Warum vor allem Symmetrie und positiv-Definitheit so bedeutend sind, wird ersichtlich, wenn man sich einmal die quadratische Form $f(x) = \frac{1}{2}x^T Ax - b^T x + c$ (A Matrix, x und b Vektoren, c skalare Konstante) ansieht. Berechnet man den Gradienten von f, so erhält man nach einigen mathematischen Umformungen: $f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b$. Im Falle der Symmetrie von A reduziert sich diese Gleichung zu $f'(x) = Ax - b$. Setzt man nun den Gradienten gleich 0, so erhalten wir genau das LGS, das wir lösen möchten. Nur wenn A positiv-definit ist, ist die Lösung ein Minimum von $f(x)$ (wäre A z. B. indefinit, ergäbe sich kein Minimum, sondern ein Sattelpunkt), so dass $Ax = b$ dadurch gelöst werden kann, dass $f(x)$ minimiert wird [17]. Genau diese Aufgabe der Minimumssuche verfolgt anschaulich gesehen das CG-Verfahren, das prinzipiell folgendermaßen funktioniert:

Im Laufe der Berechnung werden iterativ Vektorfolgen von sukzessiven Approximationen der Lösung ("Iterationswerte" x_i), Residuen, die zu diesen Approximationen gehören, und Suchrichtungen, die für "update"-Operationen an den Iterationswerten und den Residuen benötigt werden, erzeugt. Das Residuum der i-ten Iteration, das als $r_i = b - Ax_i$ definiert ist, ist dabei eine Art Maß für die Abweichung der momentanen Lösung von der tatsächlichen (interessanterweise ist dieses Residuum r_i gleichzeitig auch der negative Gradient (siehe Name des Verfahrens) von $f(x_i)$, da $-f'(x_i) = b - Ax_i$). Obwohl diese Vektorsequenzen sehr lang werden können, muss jeweils nur eine

[1]Eine Matrix A ist positiv-definit, wenn für jeden Vektor $x \neq 0$ gilt: $x^T Ax > 0$

kleine Zahl von Vektoren im Speicher behalten werden, was natürlich der Effizienz des Verfahrens zugute kommt [1].

In jeder Iteration werden zwei Skalarprodukte je zweier Vektoren ausgeführt, um "update"-Skalarwerte zu berechnen, die so definiert sind, dass die Vektorfolgen bestimmten Orthogonalitätsbedingungen genügen: jedes Residuum ist orthogonal zu allen vorhergehenden Residuen und Suchrichtungen. Wichtiger noch: jede Suchrichtung ist A-orthogonal[2] – oder *konjugiert* (siehe Name des Verfahrens) – zu allen vorhergehenden Suchrichtungen und Residuen [17]. Durch diese Bedingungen wird die "Länge der Wegstrecke" zur tatsächlichen Lösung in gewisser Weise minimiert. Bei Steepest-Descent kommt es nämlich häufig vor, dass Schritte in die gleiche Suchrichtung gemacht werden, wie schon in früheren Iterationen. Durch einen solchen "Zickzack-Kurs" entsteht eine unnötige Verlängerung des "Weges", was natürlich gleichbedeutend mit schlechterer Konvergenz ist. Dies wird durch die beim CG-Verfahren eigens berechneten Suchrichtungen und deren A-Orthogonalität verhindert.

Theoretisch wäre CG dadurch eigentlich sogar ein direktes Verfahren, das nach spätestens n Iterationen (bei n Unbekannten) die exakte Lösung erreicht haben müsste. In der Praxis geht jedoch aufgrund von sich anhäufenden Rundungsfehlern mit jeder Iteration ein gewisses Maß an Genauigkeit verloren, und die Suchrichtungen verlieren ihre A-Orthogonalität. Da die mit CG zu lösenden Probleme aber meist sehr viele Unbekannte besitzen (oft > 100000), und damit eine Zahl von n Iterationen ohnehin bei weitem zu groß wäre, fällt diese "Einschränkung" nicht ins Gewicht.

In der Konvergenzanalye beider Verfahren zeigt sich der Vorteil von CG [17]:

Steepest Descent:

$$\|e_i\|_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i \cdot \|e_0\|_A \tag{1}$$

CG:

$$\|e_i\|_A \leq 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^i \cdot \|e_0\|_A \tag{2}$$

Hierbei ist e_i der Fehler $x_{solution} - x_i$, $\|e\|_A$ die sog. Energie-Norm, definiert als $(e^T Ae)^{\frac{1}{2}}$, und κ die Konditionszahl von A, definiert als der Quotient aus dem größten und dem kleinsten Eigenwert von A: $\frac{\lambda_{max}}{\lambda_{min}}$. Wenn A positiv-definit ist, also nur positive Eigenwerte besitzt, ist dieser Quotient immer größer oder gleich 1.

Der CG-Algorithmus für einen Iterationsschritt sieht nun im genauen folgendermaßen aus [17]:

(0) Initialisierung vor der ersten Iteration, wobei x_0 ein beliebiger Startvektor sein kann (häufig wird der Nullvektor verwendet):

$$d_0 = r_0 = b - Ax_0$$

(1) Skalarwert α, der für das "update" von x_i und r_i gebraucht wird:

$$\alpha_i = \frac{{r_i}^T r_i}{{d_i}^T Ad_i}$$

(2) "update" des Iterationswertes x_i, liefert lokales Minimum in der Suchrichtung:

$$x_{i+1} = x_i + \alpha_i d_i$$

(3) "update" des Residuums r_i:

$$r_{i+1} = r_i - \alpha_i Ad_i$$

(4) Skalarwert β, der für das "update" der Suchrichtung d_i benötigt wird:

$$\beta_{i+1} = \frac{{r_{i+1}}^T r_{i+1}}{{r_i}^T r_i}$$

[2]Zwei Vektoren d_i und d_j sind A-orthogonal zueinander, wenn gilt: ${d_i}^T Ad_j = 0$, A ist Matrix

(5) “update” der Suchrichtung d_i:

$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i$$

Natürlich benötigt man nun noch ein Abbruchkriterium, um festzulegen, wann die approximierte Lösung “gut genug” ist, und die Berechnung gestoppt werden kann. Hierzu verwendet man oft das relative Residuum: $\frac{r^T r}{b^T b} \leq \epsilon$, wobei ϵ eine bestimmte Toleranzgrenze, z. B. 10^{-6}, ist.

Es bleibt schließlich noch die interessante Tatsache zu erwähnen, dass, wenn die rechte Seite b ein Eigenvektor von A ist, und der Nullvektor als Startvektor x_0 gewählt wird, das CG-Verfahren bereits nach einer einzigen Iteration die exakte Lösung gefunden hat, da in diesem Fall die erste Suchrichtung $d_0 = b - Ax_0 = b$ genau entlang dieses Eigenvektors verläuft und die Eigenvektoren von A den Achsen des zur quadratischen Form gehörenden “Paraboloids” entsprechen, sich also im gesuchten Minimum treffen.

1.2 CG mit Vorkonditionierung

Wie bereits in Abschnitt 1.1 in den Ungleichungen (1) und (2) gesehen, hängt die Konvergenzrate des CG-Verfahrens von der Größe der Konditionszahl κ der Koeffizientenmatrix A ab: je kleiner κ ist (je mehr es sich der Zahl 1 nähert), desto schneller ist die Konvergenz. Auch eine dichte “Clusterung” der Eigenwerte von A, d. h. wenn die Zahlenwerte nicht breit gestreut sind, sondern alle in einem engen Bereich liegen, beschleunigt das Verfahren.

Um nun die spektralen Eigenschaften der Koeffizientenmatrix zu verbessern, verwendet man die Technik des Vorkonditionierens. Im Prinzip wird hierbei das Gleichungssystem in ein anderes transformiert, das äquivalent ist in der Hinsicht, dass es die gleiche Lösung besitzt, jedoch günstigere spektrale Eigenschaften aufweist.

Betrachten wir eine symmetrische und positiv-definite Matrix M, die A approximiert, aber leichter zu invertieren ist. Wir können $Ax = b$ indirekt dadurch lösen, dass wir das System

$$M^{-1}Ax = M^{-1}b \tag{3}$$

lösen. Wenn nun $\kappa(M^{-1}A) \ll \kappa(A)$ oder wenn die Eigenwerte von $M^{-1}A$ besser “geclustert” sind als die von A, können wir die Gleichung (3) iterativ schneller lösen als das ursprüngliche Problem [17].

Auf den ersten Blick problematisch ist nun allerdings die Tatsache, dass die Matrix $M^{-1}A$ im allgemeinen nicht symmetrisch und positiv-definit ist, auch wenn es M und A sind. Diese Schwierigkeit kann jedoch umgangen werden, da für jede symmetrische positiv-definite Matrix M eine Matrix E existiert mit der Eigenschaft, dass $EE^T = M$. Die Matrizen $M^{-1}A$ und $E^{-1}AE^{-T}$ besitzen dann die gleichen Eigenwerte [17]. Das LGS $Ax = b$ kann damit in das Problem

$$E^{-1}AE^{-T}\hat{x} = E^{-1}b, \quad \hat{x} = E^T x \tag{4}$$

transformiert werden, das zunächst für $\hat{x}$ und dann für x gelöst wird. Da $E^{-1}AE^{-T}$ symmetrisch und positiv-definit ist, kann $\hat{x}$ mithilfe von CG ermittelt werden.

In dem auf diese Weise entstehenden sog. “Transformierten vorkonditionierten CG-Verfahren” lässt sich nun durch einige Variablensubstitutionen (siehe [17]) die Matrix E eliminieren, so dass das “Untransformierte vorkonditionierte CG-Verfahren” entsteht, bei dem nur die Matrix M^{-1} benötigt wird, und E somit überhaupt nicht berechnet werden muss. Die einzelnen Schritte sehen beim “Untransformierten vorkonditionierten CG-Verfahren” dann folgendermaßen aus [17]:

$$r_0 = b - Ax_0$$
$$d_0 = M^{-1}r_0$$

$$\alpha_i = \frac{{r_i}^T M^{-1} r_i}{{d_i}^T A d_i}$$

$$x_{i+1} = x_i + \alpha_i d_i$$

$$r_{i+1} = r_i - \alpha_i A d_i$$

$$\beta_{i+1} = \frac{{r_{i+1}}^T M^{-1} r_{i+1}}{{r_i}^T M^{-1} r_i}$$

$$d_{i+1} = M^{-1} r_{i+1} + \beta_{i+1} d_i$$

Man erkennt, dass die Vorkonditionierung für den CG-Algorithmus lediglich eine Multiplikation von M^{-1} mit dem aktualisierten Residuumsvektor r_{i+1} pro Iteration mehr bedeutet ($M^{-1} r_i$ ist ja jeweils immer bereits in der vorangegangenen Iteration berechnet worden). Wichtig hierbei ist, dass es nicht nötig ist, M oder M^{-1} explizit zu berechnen, sondern nur, dass der Effekt des "Anwendens" von M^{-1} auf einen Vektor berechnet werden kann.

Im Grunde gibt es zwei Arten von Vorkonditionierern: entweder eine Matrix M, die A approximiert und für die die Lösung eines Gleichungssystems (z. B. $Ms = r$) leichter ist als mit A – *Euclid* fällt unter diese Kategorie – oder eine Matrix M^{-1}, die A^{-1} approximiert, so dass nur eine Multiplikation mit M^{-1} nötig ist – *ParaSails* ist ein Beispiel dafür.

Da die Verwendung eines Vorkonditionierers in einem iterativen Verfahren immer mit zusätzlichen Kosten verbunden ist, sowohl zu Beginn für das Aufstellen der Matrix ("setup"), als auch in jeder Iteration für die Anwendung ("apply"), bleibt es ein allgemeines Problem, einen Vorkonditionierer zu finden, der einerseits genau genug ist, um die Konvergenz deutlich zu verbessern, andererseits aber gleichzeitig keine übermäßig hohen setup-Kosten und "Anwendungskosten" (siehe auch "operation count", Abschnitt 4) aufweist, damit letztlich insgesamt noch eine sichtbare zeitliche Verbesserung gegenüber der nicht-vorkonditionierten Berechnung zu erkennen ist. Weil der "setup" nur einmal ganz am Anfang von womöglich mehreren Lösungsberechnungen mit der gleichen Matrix und lediglich verschiedenen rechten Seiten (wie in unserem Fall beim "3D-Kopfproblem", siehe Abschnitt 8) nötig ist, fällt dieser Teil der Mehrkosten jedoch meistens nur bei kleineren Gleichungssystemen mit relativ kurzen Lösungszeiten prozentual gesehen ins Gewicht.

2 Die *hypre*-Programmbibliothek

Die *hypre*[3]-Programmbibliothek (die Programmiersprache ist im wesentlichen C, ganz wenig C++), deren Version 1.6.0 aus dem Jahr 2001 für diese Arbeit zum Einsatz kam, wurde vom Lawrence Livermore National Laboratory der University of California in Zusammenarbeit mit dem U.S.-Energie-Ministerium entwickelt und dient zur Lösung von großen dünn-besetzten ("sparse") LGS auf hochgradig parallelen Rechnern [9]. Da die Implementierung der für unsere Tests relevanten Verfahren jedoch genauso deren Verwendung im sequenziellen Fall zulässt, war das *hypre*-Software-Paket auch für unsere Zwecke geeignet.

Für eine möglichst effiziente Lösungsberechnung bietet *hypre* dem Benutzer sowohl eine Auswahl verschiedener Arten von Vorkonditionierern ("preconditioner") – darunter auch die bereits erwähnten *Euclid*- und *ParaSails*-Verfahren – an, als auch mehrere der am häufigsten verwendeten Krylov-basierten iterativen Lösungsmethoden ("solver"), wie z. B. geometrische und algebraische Mehrgitter-Verfahren und das bei unseren Tests eingesetzte CG-Verfahren [9].

Um dem Anwender soviel Arbeit wie möglich zu ersparen – und wohl auch, um die Performance zu steigern – gibt es bei *hypre* verschiedene sog. *konzeptionelle Interfaces*, die im Grunde jeweils eine bestimmte Gattung von modellbasierten Berechnungsgittern, wie sie gerade bei Diskretisierungsvorgängen eine große Rolle spielen, unterstützen. Diese Interfaces sollen die Vorstellung repräsentieren, die Anwendungsentwickler intuitiv von ihrem linearen Problem besitzen. Auf diese Weise muss sich der Benutzer nicht mit komplizierten Datenstrukturen für dünn-besetzte Matrizen abgeben, sondern *hypre* übernimmt deren Erstellung aufgrund der vom Benutzer spezifizierten Interface-Daten [9]. Beispielsweise werden bei Anwendungen, die strukturierte Gitter verwenden, die linearen Probleme typischerweise durch eine "stencil"-Sichtweise (z. B. "five-point-stencil" beim 2D-Poisson-Problem, siehe Abschnitt 5) betrachtet und dabei eher weniger an die

[3] *hypre* ist abgeleitet von "high performance preconditioners"

daraus resultierende dünn-besetzte Matrix gedacht. Insgesamt gibt es bei *hypre* momentan vier verschiedene Schnittstellen [9]:

- *Structured-Grid System Interface (Struct)*:
  Diese Schnittstelle ist für Anwendungen geeignet, deren Gitter aus rechteckigen Gitterzellen mit einem festgelegten "stencil"-Muster der Nichtnull-Werte bezüglich jedes Gitterpunktes besteht (siehe "five-point-stencil" beim 2D-Poisson-Problem). Es wird dabei nur eine einzige Unbekannte pro Gitterpunkt unterstützt.

- *Semi-Structured-Grid System Interface (SStruct)*:
  Dies ist für Anwendungen vorgesehen, deren Gitter größtenteils strukturiert ist, jedoch auch einige unstrukturierte Merkmale aufweist, wie beispielsweise bei blockstrukturierten Gittern. Hier werden mehrere Unbekannte pro Gitterpunkt unterstützt.

- *Finite Element Interface (FEI)*:
  Dieses interface richtet sich an Benutzer, die ihr LGS aus einer Finite-Elemente-Diskretisierung erstellen. Es bietet dabei typische Datenstrukturen dieser Methode an.

- *Linear-Algebraic System Interface (IJ)*:
  Dies ist das traditionelle "Lineare-Algebra-Interface". Es beinhaltet im Vergleich zu den anderen Schnittstellen mehr Arbeit für den Benutzer, kann dabei aber als letzter Ausweg für Anwendungen dienen, für die die anderen Gitter-basierten Interfaces nicht geeignet sind. Es ist für jeden Gittertyp verwendbar und deshalb sozusagen der kleinste gemeinsame Nenner, um LGS mit *hypre* zu spezifizieren, jedoch können nur allgemeine "solver" für dünn-besetzte Matrizen (z. B. CG) zum Einsatz kommen und keine spezialisierten, die spezifischere Probleminformationen benötigen (beispielsweise kann der GMG (geometric multigrid)-Algorithmus nur im dafür vorgesehenen Interface benutzt werden).

Die Schnittstelle unserer Wahl ist das letztgenannte *IJ-Interface*, da es das allgemeinste und universellste aller Interfaces darstellt, für unsere betrachteten Probleme und die dafür eingesetzten "solver" und "preconditioner" gut verwendbar ist, und außerdem die anderen Interfaces zum Teil noch gar nicht komplett implementiert sind.

Das *IJ-interface* besteht nun seinerseits wieder aus zwei "Unter-Interfaces": dem *IJ-Matrix-Interface* und dem *IJ-Vektor-Interface* [9].

Das *IJ-Matrix-Interface* dient – wie der Name schon sagt – dem Aufstellen der Koeffizientenmatrix des zu lösenden linearen Problems. Die Dateneingabe erfolgt hierbei zeilenweise, d. h. nachdem die Größe der Matrix festgelegt und ein leeres Matrixobjekt (bestimmte Datenstruktur) mithilfe der Funktionen `MatrixCreate` und `MatrixInitialize` erzeugt worden ist, werden schrittweise die einzelnen Zeilen der Matrix spezifiziert (hierzu werden jeweils zwei Arrays – eines für die Werte der Einträge und eines für deren Positionen in der Zeile – verwendet) und mithilfe der `MatrixSetValues`-Routine in das Matrix-Objekt eingefügt. Am Ende wird durch die `MatrixAssemble`-Funktion die Matrixerstellung abgeschlossen und die Matrix zur Nutzung "freigegeben".

Das *IJ-Vektor-Interface* dient der Erstellung von Vektor-Objekten, von denen zwei für die Lösung eines LGS nötig sind: eines für die rechte Seite b und eines für den Vektor der Unbekannten x. Auch hier gibt es eine `Create`- und eine `Initialize`-Routine, die einen leeren Vektor einer bestimmten Größe erstellen. Dieser wird dann durch die `VectorSetValues`-Funktion mit den gewünschten Werten der rechten Seite bzw. den Startwerten des Unbekannten-Vektors gefüllt und abschließend ebenfalls mithilfe einer `Assemble`-Routine "freigegeben".

Nachdem wir die zu unserem Problem gehörende Koeffizientenmatrix A und die beiden Vektoren x und b erstellt haben, steht nun die Auswahl des Lösungsverfahren – in unserem Fall CG – und des eventuell darauf anzuwendenden Vorkonditionierers – in unserem Fall *Euclid* oder *ParaSails* – an. Normalerweise wird der Vorkonditionierer vor der eigentlichen Erstellung des "solvers" spezifiziert, um dann an diesen im Rahmen des "solver-setup" übergeben zu werden. Es ergeben sich zumeist folgende Schritte:

Zunächst wird mit einer `Create`-Routine ein (noch leeres) "solver"-Objekt (des gewünschten "solvers") erstellt (dies ist im wesentlichen eine große Datenstruktur). Nachfolgend können durch `Set`-Aufrufe verschiedene den "solver" betreffende Parameter, wie z. B. die maximal zulässige Iterationenanzahl oder die Toleranzgrenze ϵ für das Abbruchkriterium, eingestellt werden.

Möchte man eine Vorkonditionierung nützen, schließt sich die Erstellung des Vorkonditionierer-Objekts des gewünschten Vorkonditionierers mithilfe einer `Create`-Routine an. Auch hier können nun wieder spezielle den Vorkonditionierer betreffende Parameter und Optionen gesetzt werden (`Set`-Funktionen), die z. B. bestimmte "Genauigkeits-levels" (siehe *Euclid* und *ParaSails*, Abschnitt 3) oder die Frage nach der Ausgabe einer Statistik am Ende betreffen. Ist dies alles geschehen, wird das Vorkonditionierer-Objekt durch eine `SetPrecond`-Routine sozusagen mit dem "solver"-Objekt vereinigt, um dadurch ein gemeinsames Objekt zu erhalten, das alle bisher festgelegten Daten enthält. Dieser Vorgang entfällt natürlich, wenn auf eine Vorkonditionierung verzichtet wird.

Erst jetzt erfolgt die eigentliche Aufstellung des "solvers" mithilfe der `setup`-Routine. Dazu werden nun auch die bereits erstellten Matrix- und Vektorobjekte, die ja die eigentlichen Daten des zu lösenden Problems enthalten, übergeben. Auch der setup des Vorkonditionierers, also im wesentlichen das Aufstellen einer bestimmten Matrix, findet erst innerhalb dieser Routine statt.

Damit ist alles bereit, um die Lösung des Problems anzugehen. Es muss nur noch die `solve`-Funktion aufgerufen werden, und der Lösungsvorgang kann beginnen. Nach dessen erfolgreicher Beendigung, d. h. wenn das gewählte Abbruchkriterium erreicht ist, kann man sich abschließend mithilfe einer `getValue`-Funktion die Lösungsdaten aus dem Vektorobjekt des Unbekanntenvektors herausholen und diese je nach Anwendung eventuell gleich weiter verarbeiten. Außerdem werden am Ende je nach vorheriger Einstellung noch Statistik-Daten, z. B. die Anzahl der benötigten Iterationen, ausgegeben [9].

Für weitere und genauere Informationen zur *hypre*-Bibliothek, insbesondere zu den verschiedenen Interfaces, siehe auch [5] und [6]. Allgemein wird auch ein Blick auf [7] empfohlen.

3 Die *hypre*-Vorkonditionierer *ParaSails* und *Euclid*

Nach der Beschreibung der allgemeinen Eigenschaften der *hypre*-Bibliothek und der grundsätzlichen Vorgehensweise bei der Anwendung, sollen nun die beiden in dieser Arbeit getesteten Vorkonditionierer genauer betrachtet werden. Dazu werden jeweils sowohl der mathematische Hintergrund, als auch die spezielle Situation bei *hypre* – beispielsweise die Auswirkungen bestimmter Parametereinstellungen – erläutert.

3.1 ParaSails

ParaSails ist Teil der *hypre*-Programmbibliothek und stellt die Implementierung eines "sparse approximate inverse" (SPAI)-Vorkonditionierers dar, der eine dünn-besetzte und der Inversen der Koeffizientenmatrix A "ähnliche" Matrix M konstruiert, indem $I - MA$ in der Frobenius-Norm[4] durch least-squares-Berechnungen minimiert wird [4].

Wichtig ist vor der Minimierung die geeignete Wahl eines "sparsity pattern" ("Besetztheitsmuster") für M, das angibt, wo überall Nichtnull-Einträge zugelassen sind. Da ein effektives "sparsity pattern" von M im allgemeinen a priori unbekannt ist, versucht der ursprüngliche SPAI-Algorithmus von Grote und Huckle (1997), siehe auch [2], die für eine gute Approximierung vielversprechendsten Einträge dynamisch festzulegen. Dazu beginnt der Algorithmus mit einem Diagonalmuster und verdichtet dann schrittweise das "pattern" bis der Wert von $\|I - MA\|_F$ kleiner als eine vorher gewählte Toleranzgrenze, die sich aus einem Parameter ϵ ergibt, ist. Man spricht deshalb in diesem Zusammenhang auch von einem SPAI(ϵ)-Verfahren, wobei ϵ die Genauigkeit und die Menge an "fill-in" von M kontrolliert: je größer ϵ, desto dünn-besetzter, aber auch ungenauer wird die approximative Inverse von A und umgekehrt.

Trotzdem können auch durch die Wahl eines bereits vor der Minimierung statisch festgelegten "a priori sparsity pattern" für M gute Ergebnisse erzielt werden und dabei die Berechnungskosten

[4]Frobenius-Norm einer Matrix A: $\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} {a_{ij}}^2}$

(setup) stark reduziert werden, da die aufwändige Suche nach einem effektiven Muster entfällt. Mögliche Kandidaten für das "pattern" sind hierbei v. a. die Besetzungsmuster der Potenzen der Matrix A. Innerhalb dieser Verfahren mit "a priori pattern" für M werden zwei Spezialfälle noch gesondert bezeichnet [2]:

- SPAI-0: hier hat M lediglich ein Diagonalmuster
- SPAI-1: hier ist das "sparsity pattern" von M mit dem von A identisch

Der Vorteil an den SPAI-Verfahren besteht darin, dass die Matrix M direkt berechnet wird (d. h. eigentlich bereits der Matrix M^{-1} in der Definition des Vorkonditionierens entspricht) und dadurch die "Anwendung"("apply") des Vorkonditionierers nur Matrix-Vektor-Multiplikationen, die zudem auch leicht zu parallelisieren sind, beinhaltet und nicht das Lösen irgendeines auf einer oberen oder unteren Dreiecksmatrix basierenden Gleichungssystems erfordert. Außerdem führt die spezielle Definition der Frobenius-Norm natürlicherweise zu inhärentem Parallelismus, da die n Zeilen ${m_k}^T$ von M unabhängig voneinander berechnet werden können. Es gilt nämlich:

$$\|I - MA\|_F^2 = \sum_{k=1}^{n} \|{e_k}^T - {m_k}^T A\|_2^2 = \sum_{k=1}^{n} \|A^T m_k - e_k\|_2^2 \quad , \tag{5}$$

wobei e_k den k-ten Einheitsvektor bezeichnet. Somit teilt sich die Lösung der Minimierung von (5) in n unabhängige least-squares-Probleme für die m_k auf: $\min_{m_k} \|A^T m_k - e_k\|_2^2$ (siehe auch [2] und [3]). Da A und M jeweils dünn-besetzt ("sparse") sind, sind diese Probleme klein.

SPAI-Verfahren sind also – im Gegensatz zu anderen Vorkonditionierungsverfahren, wie beispielsweise die inkomplette LU-Zerlegung, auf der *Euclid* basiert (siehe Abschnitt 3.2) – geradezu prädestiniert für die Anwendung auf Parallelrechnern und werden dort wohl die besten Performance-Ergebnisse liefern, auch wenn sie – wie wir in unseren Tests noch sehen werden – im sequenziellen Fall eventuell nicht so gut abschneiden.

Auch die *ParaSails*-Implementierung bei *hypre* benützt "a priori sparsity patterns" für M. Diese "patterns" sind dabei die Muster von Potenzen von durch "sparsification" aus A entstandenen dünn-besetzten Matrizen. Nach der Erstellung von M wird zudem noch eine "post-filtering"-Technik eingesetzt, wobei kleine Werte in M gestrichen werden, um die Kosten der "Anwendung" des Vorkonditionierers zu reduzieren [4].

Zunächst wird das "sparsity-pattern" festgelegt. Dazu gibt es die beiden Parameter m (ganzzahlig, $m \geq 0$) und *thresh* (reell, $thresh \geq 0$). Sei eine Matrix A gegeben, so ergibt sich das "sparsity pattern" des Vorkonditionierers M aus dem "sparsity pattern" von $\tilde{A}^m$, wobei $\tilde{A}$ eine Binärmatrix ist, definiert als [4]:

$$\tilde{A}_{ij} = \begin{cases} 1 & : \quad i = j \ \vee \ |(D^{-\frac{1}{2}} A D^{-\frac{1}{2}})_{ij}| \ > \ thresh \\ 0 & : \quad \text{sonst} \end{cases}$$

mit der Diagonalmatrix D, definiert als:

$$D_{ii} = \begin{cases} |A_{ii}| & : \quad |A_{ii}| \ > \ 0 \\ 1 & : \quad \text{sonst} \end{cases}$$

$(D^{-\frac{1}{2}})_{ii}$ bedeutet hier $\dfrac{1}{\sqrt{d_{ii}}}$, so dass sich $|(D^{-\frac{1}{2}} A D^{-\frac{1}{2}})_{ij}|$ deshalb auch (lesbarer) als $\dfrac{|a_{ij}|}{\sqrt{|a_{ii} a_{jj}|}}$ schreiben lässt. Dieser Quotient ist also in gewisser Weise ein Maß für das "Gewicht" eines bestimmten Koeffizienten im Vergleich zu den beiden Elementen auf der Hauptdiagonalen in der betreffenden Zeile bzw. Spalte.
Je größer m und je kleiner der Schwellwert *thresh*, desto dichter besetzt wird die Matrix M und umgekehrt.

Die Minimierung erfolgt nun für symmetrische und nicht-symmetrische Matrizen A auf verschiedene Art und Weise (mithilfe des Parameters *symmetry* lässt sich der jeweils vorliegende Fall

einstellen). Ist A nicht symmetrisch, wird die "approximate inverse" M durch die Minimierung von $\|I - MA\|_F$ berechnet, wobei das vorher bestimmte "sparsity pattern" von $\tilde{A}^m$ als Grundlage dient. Eine Arbeitserleichterung ergibt sich, wenn A symmetrisch ist (was bei unseren Tests der Fall ist), denn dann wird prinzipiell nur ein "approximate inverse"-Faktor G in der Gestalt einer unteren Dreiecksmatrix durch die Minimierung von $\|I - GL\|_F$ berechnet, wobei $A = LL^T$ die Cholesky-Faktorisierung von A ist und G das gleiche "pattern" besitzt wie der untere Dreiecksteil der Matrix $\tilde{A}^m$ [4]. Da $G^T GLL^T = G^T GA \approx I$, ergäbe das Produkt $G^T G$ die Vorkonditionierer-Matrix M. Natürlich stellt sich nun die Frage nach dem Cholesky-Faktor L, es lässt sich jedoch zeigen (siehe [3] und insbesondere [12]), dass für die Minimierungsaufgabe die explizite Berechnung von L nicht nötig ist [5](dies würde ja auch einen beträchtlichen Arbeitsaufwand bedeuten, der die Einsparungen sofort wieder zunichte machen würde).

Wie bereits angedeutet, besteht nun, nachdem die Werte von M (bzw. G) berechnet worden sind, die Möglichkeit, kleine Einträge von M "fallenzulassen", um die Kosten des Multiplizierens mit M ("apply" des Vorkonditionierers) zu reduzieren. Einträge in M werden gestrichen, wenn sie kleiner als ein bestimmter Schwellwert, der sog. Filterwert, sind.

Insgesamt besitzt *ParaSails* also drei wichtige Parameter, um die Genauigkeit und die Kosten des Verfahrens einzustellen [4]:

- *thresh* ($\in \mathbb{R}_0^+$): entspricht dem oben genannten Wert *thresh*; Schwellwert, um das Matrix-"pattern" *vor* dem Minimierungsvorgang auszudünnen ("sparsification"). Ab einer bestimmten, vom jeweiligen Problem abhängigen Größe von *thresh*, ergibt sich das SPAI-0-Verfahren.

- *nlevels* ($\in \mathbb{N}_0^+$): m in $\tilde{A}^m$ entspricht *nlevel* + 1, d. h. bei *nlevel* = 0 (und *thresh* = 0) ist ist das "sparsity pattern" von M das gleiche wie das von A: es ergibt sich SPAI-1. Sinnvolle Werte für *nlevels* liegen meist zwischen 0 und 10.

- *filter* ($\in \mathbb{R}_0^+$): oben genannter Filterwert

Sowohl beim *filter*-, als auch beim *thresh*-Parameter besteht zudem die Möglichkeit, negative Werte zwischen -1 und 0 einzusetzen. In diesem Fall gibt der Betrag dieser Werte den Anteil der Nichtnull-Einträge an , die gestrichen werden sollen (z. B. *filter* = -0.9: 90% der Einträge werden "fallengelassen") [4]. Der zur jeweiligen Prozentangabe gehörende Absolutwert wird dann automatisch bestimmt und kann in den Statistikangaben eingesehen werden.

Wie bereits erwähnt, führen niedrige Werte von *thresh* und höhere Werte von *nlevels* – da die Matrix M dadurch dichter besetzt ist – zu genaueren [6], aber auch teureren Vorkonditionierern, wobei größere Werte von *nlevels* vor allem auch in höheren setup-Kosten resultieren. Allgemein bedeuten dichter besetzte Vorkonditionierer-Matrizen mehr Berechnungsaufwand pro Iteration, dafür ergeben sich insgesamt jedoch meist weniger Iterationen, so dass ein guter Kompromiss zwischen einer akzeptablen Genauigkeit von M einerseits und einem sich in Grenzen haltenden Rechenaufwand pro Iteration andererseits gefunden werden muss. Sind dabei – wie im Falle von *ParaSails* – mehrere Parameter von Bedeutung, ist auf jeden Fall zu Beginn eine Parameterstudie (wie später beim Kopfproblem, Abschnitt 8.2) mit dem Ziel der Optimierung nach der Gesamtzeit der Lösungsberechnung bzw. der insgesamten Anzahl an benötigten elementaren Rechenoperationen (Addition und Multiplikation, siehe "operation count", Abschnitt 4) notwendig. Die dafür benötigten Daten zur Anzahl der Nichtnull-Einträge der jeweils erhaltenen Matrix M finden sich in den Statistik-Angaben von *ParaSails*.

3.2 Euclid

Der *Euclid*-Vorkonditionierer, dessen Implementierung zwar ebenfalls Teil der *hypre*-Bibliothek ist, jedoch unabhängig vom *hypre*-Projekt entwickelt wurde (David Hysom und Alex Pothen, [11]), basiert auf einer inkompletten LU-Zerlegung (Faktorisierung) der Koeffizientenmatrix – eine beliebte, effiziente und relativ robuste Technik, um die Konvergenz von Krylov-Lösungsverfahren für LGS zu verbessern.

[5]Die Minimierungsaufgabe verändert jedoch dadurch ihre Gestalt; $\min\|I - GL\|_F$ wird in *der* Form dann nicht mehr gelöst.

[6]Die Inverse einer dünn-besetzten Matrix ist in der Regel voll-besetzt.

Die Faktorisierung einer Matrix A wird als inkomplett bezeichnet, wenn während dieses Prozesses bestimmte sog. 'fill"-Elemente – Nichtnull-Einträge, die im Laufe der Faktorisierung an solchen Positionen der beiden Faktor-Matrizen L und U erzeugt werden, wo die ursprüngliche Matrix einen Nullwert besitzt – ignoriert bzw. unterdrückt werden [1]. Daraus resultiert dann die Gleichung $A = LU + R$, wobei L eine untere und U eine obere Dreiecksmatrix ist und R eine "Restmatrix". Das Produkt LU stellt nun die (in der faktorisierten Form gegebene) Vorkonditionierungsmatrix M dar, deren Effektivität im wesentlichen davon abhängt, wie gut M^{-1} die Matrix A^{-1} approximiert.

Gewöhnlich wird bei der ILU-Faktorisierung eine Menge $\mathcal{S}$ von Matrixpositionen definiert und alle nicht in $\mathcal{S}$ enthaltenen Positionen während des Faktorisierungsvorgangs gleich Null gehalten. $\mathcal{S}$ wird dabei normalerweise so gewählt, dass alle Positionen (i, j), für die $a_{ij} \neq 0$, in ihr enthalten sind. Eine Position in L bzw. U, die in A den Wert Null besitzt, nicht aber in der genauen (also "kompletten") Faktorisierung von A, nennt man "fill"-Position. Ist diese außerhalb von $\mathcal{S}$, wird der dort eigentlich vorgesehene "fill"-Wert fallengelassen (d. h. der Wert bleibt 0) [1].

Oft – so auch bei *Euclid* – wird die Strategie verfolgt, "fill-in" bis zu einem bestimmten *level* zu akzeptieren. Hierzu wird jeder "fill"-Position (i, j), die im Laufe der Faktorisierung einen Nichtnull-Eintrag bekommen soll, ein sog. *fill-level*-Wert zugeordnet. Ein Element wird dabei vom *level* $k+1$ bezeichnet, wenn seine Entstehung in einem bestimmten Schritt des Faktorisierungsvorgangs durch Elemente hervorgerufen wird, von denen mindestens eines vom *level* k ist. Den häufigen Fall, dass $\mathcal{S}$ so gewählt wird, dass es genau mit der Menge an Nichtnull-Positionen in A zusammenfällt, bezeichnet man dabei als *ILU(0)*-Faktorisierung (*level* $= 0$). Daher wird das erste *fill-level* immer von Nichtnull-Elementen der ursprünglichen Matrix "verursacht" [1], [11].

Etwas problematisch ist die Tatsache, dass inkomplette Faktorisierungsvorgänge unter bestimmten Umständen ohne Erfolg abbrechen oder in indefiniten Matrizen LU resultieren können, auch wenn die Existenz einer *vollständigen* Faktorisierung der gleichen Matrix A garantiert ist [1]. Meijerink und Van der Vorst bewiesen allerdings [13], dass die Existenz einer inkompletten Faktorisierung für jede Wahl von $\mathcal{S}$ garantiert ist, und diese im Falle einer symmetrischen positiv-definiten (s.p.d.) Koeffizientenmatrix A auch eine s.p.d. Matrix LU liefert, wenn A eine sog. *M-Matrix*[7] ist.

Eine wichtige Überlegung für ILU-Vorkonditionierer sind auch die Kosten des Faktorisierungsvorgangs (setup), denn auch wenn die inkomplette Zerlegung existiert, ist die Zahl der zu ihrer Herstellung erforderlichen Operationen mindestens ebenso groß wie für die Lösung eines Gleichungssystems mit einer so entstandenen Matrix, so dass die Kosten genauso groß sein können wie die von einer oder mehrerer Iterationen der zugrundeliegenden iterativen Methode [1].

Ist die Vorkonditionierungsmatrix $M = LU$ in der "setup-Phase" erstellt, kann die Anwendung des Vorkonditionierers ("apply-Phase") im iterativen Lösungsverfahren erfolgen. Wie bereits (in Abschnitt 1.2) erwähnt, ist es dazu nicht nötig, die Inverse M^{-1} explizit zu berechnen, sondern lediglich die "Wirkung" deren Multiplikation mit einem Vektor, beispielsweise dem Residuumsvektor r beim CG-Verfahren, zu "simulieren". Dazu wird das Gleichungssystem

$$Ms = r \quad \equiv \quad LUs = r$$

betrachtet, wobei r der (vorgegebene) vorzukonditionierende und s der (zu berechnende) vorkonditionierte Vektor ist. Dieses Gleichungssystem wird in zwei Untersysteme aufgeteilt [11]:

$$(1) \quad Lw = r \text{ , } \quad \text{wobei } w \text{ ein "Hilfsvektor" ist}$$

Dies wird durch Vorwärtssubstitution gelöst.

$$(2) \quad Us = w$$

Dies wird durch Rückwärtssubstitution gelöst.

Damit hat man nun indirekt $s = M^{-1}r$ berechnet.

[7] Eine Matrix $T \in R^{n,n}$ heißt *M-Matrix*, falls T invertierbar ist, und $a_{ii} \geq 0$, $a_{ij} \leq 0$ $(i \neq j)$ und $T^{-1} \geq 0$, wobei "$\geq$" zuletzt komponentenweise zu verstehen ist.

Der wichtigste und für unsere Zwecke einzig relevante Parameter bei *hypre-Euclid* ist das *level k* ($k \in N_0^+$), das dem oben genannten *fill-level* entspricht. Je größer k ist, desto dichter besetzt und damit genauer wird die Vorkonditionierungsmatrix M (sinnvolle Werte liegen – wie wir auch noch in unseren Tests sehen werden – meist zwischen 0 und 10). Bei einem *level* von 0 besitzt die sog. *fill-Matrix* $F = L + U - I$ (I ist die Einheitsmatrix), deren jeweilige Anzahl an Nichtnull-Einträgen in den Statistik-Daten von *Euclid* eingesehen werden kann, die gleiche Anzahl an "Nichtnullen" wie A ($\rightarrow$ *ILU(0)*).

Es stellt sich natürlich auch hier die Aufgabe, je nach Problem einen bezüglich der Lösezeit optimalen *level*-Wert zu finden, was jedoch bei nur einem Parameter im Vergleich zu *ParaSails* deutlich übersichtlicher und leichter ist. Allgemein gilt die Faustregel, dass sich für 2D-Konvektions-Diffusions-Probleme und ähnliche Aufgabenstellungen (siehe z. B. 2D-Poisson-Problem, Abschnitt 5) die schnellste Lösungszeit für *level*-Werte zwischen 4 und 8 ergibt, während bei 3D-Problemen (z. B. 3D-Poisson-Problem, siehe "Kopfproblem", Abschnitt 8) das beste Ergebnis mit *level* 1 erreicht wird [11].

4 Operation Count

Um die Effizienz und Qualität eines iterativen Lösungsverfahrens bzw. eines Vorkonditionierers bewerten zu können, genügt es natürlich nicht, nur die auftretende Anzahl von Iterationen bis zum Erreichen der gewählten Fehlertoleranz zu betrachten. So kann ein Verfahren mit einer sehr geringen Anzahl von Iterationen, aber einem großen Berechnungsaufwand innerhalb eines Iterationsschrittes, insgesamt unter Umständen deutlich mehr Rechenaufwand und damit auch Zeitaufwand – und dies ist ja letztlich immer der entscheidende Faktor bei der Bewertung eines Algorithmus, der auf einem Rechner implementiert werden soll – bedeuten als ein Verfahren, das zwar insgesamt mehr Iterationen benötigt, diese jedoch jeweils relativ günstig auszuführen sind.

Beispielsweise würde das CG-Verfahren mit einer Vorkonditionierungsmatrix der Gestalt A^{-1} nur eine einzige Iteration benötigen, der Aufwand, um die Matrix aufzustellen wäre jedoch enorm, da dies bereits die Lösung des gesamten Problems darstellen würde.

Aufgrund dieser Überlegungen ist es erforderlich, eine andere Maßzahl als Bewertungskriterium zu wählen. Hierbei bietet sich die insgesamt bis zum Erreichen der gewünschten Genauigkeit benötigte Anzahl an elementaren arithmetischen Operationen – Additionen und Multiplikationen – an.

Im folgenden soll nun ein solcher "operation count" betrachtet werden: zunächst für das "reine" CG-Verfahren, um dann darauf aufbauend den Fall der Vorkonditionierung mit *Euclid* und *ParaSails* mit einzubeziehen. Gerade im Hinblick auf eine Parameterstudie, um für *Euclid* und v. a. für *ParaSails* die optimalen Parameter bezüglich des Rechenaufwandes für ein bestimmtes zu lösendes Problem (z. B. das "Kopfproblem", Abschnitt 8) zu finden, sind die dadurch gewonnenen Daten unerlässlich.

4.1 Operation Count ohne Vorkonditionierung

Beim CG-Verfahren ohne Vorkonditionierung sind zwei Parameter für den Berechnungsaufwand entscheidend: die Anzahl der Nichtnull-Einträge in der Koeffizientenmatrix A, $nz(A)$, und die Anzahl der Zeilen (Spalten) von A (also die Anzahl der Unbekannten), n.

Zu Beginn steht zunächst die Berechnung von $r_0 (= d_0)$ an: $r_0 = b - Ax_0$
Hierbei sind für die Matrix-Vektor-Multiplikation $nz(A)$ Multiplikationen und $nz(A) - n$ Additionen erforderlich. Dazu kommen noch n Subtraktionen ($\equiv$ Additionen), so dass für die Initialisierung insgesamt $nz(A)$ Multiplikationen und $nz(A)$ Additionen, also $2nz(A)$ Operationen anfallen.

Betrachten wir nun den eigentlichen Iterationsschritt, so müssen folgende "Komponenten" berechnet werden (vergleiche dazu auch die Beschreibung des CG-Verfahrens, Abschnitt 1):

Ad_i: Matrix-Vektor-Multiplikation; benötigt $nz(A)$ Multiplikationen und $nz(A) - n$ Additionen

${d_i}^T Ad_i$: Skalarprodukt zweier Vektoren (Ad_i ist bereits berechnet); benötigt n Multiplikationen und $n-1$ Additionen

α_i: Division (Dividend und Divisor sind bereits berechnet); benötigt eine Multiplikation

x_{i+1}: Multiplikation eines Vektors mit einer Konstanten und Vektor-Vektor-Addition; benötigt n Multiplikationen und n Additionen

r_{i+1}: Multiplikation eines Vektors mit einer Konstanten und Vektor-Vektor-Subtraktion ($\equiv$ Addition); benötigt n Multiplikationen und n Additionen

${r_{i+1}}^T r_{i+1}$: Skalarprodukt zweier Vektoren; benötigt n Multiplikationen und $n-1$ Additionen

β_{i+1}: Division (Dividend und Divisor sind bereits berechnet); benötigt eine Multiplikation

d_{i+1}: Multiplikation eines Vektors mit einer Konstanten und Vektor-Vektor-Addition; benötigt n Multiplikationen und n Additionen

Zählen wir diese Operationen zusammen, so erhalten wir pro Iteration $nz(A) + 5n + 2$ Multiplikationen und $nz(A) + 4n - 2$ Additionen, also $2nz(A) + 9n$ Operationen.

Schließlich ergibt sich zusammen mit den Initialisierungskosten folgende Formel für die Anzahl an insgesamt erforderlichen elementaren Operationen bei einer Lösungsberechnung mithilfe des CG-Verfahrens, die k Iterationen benötigt:

$$\text{Ops(CG)} = 2nz(A) + k \cdot (2nz(A) + 9n)$$

4.2 Operation Count mit Vorkonditionierung

Abgesehen von einer Kostensteigerung durch das Aufstellen ("setup") der Vorkonditionierungsmatrix M vor Beginn der eigentlichen iterativen Lösungsberechnung, schlägt sich die Vorkonditionierung einer Koeffizientenmatrix A natürlich hauptsächlich durch einen erhöhten Berechnungsaufwand in Form von zusätzlichen elementaren Operationen pro Iteration ("apply") nieder. Wie bereits gesehen (in Abschnitt 1.2), bedeutet das Vorkonditionieren beim CG-Verfahren pro Iterationsschritt lediglich eine "Anwendung" von M^{-1} auf den Vektor r_{i+1}, so dass der Mehraufwand gegenüber dem "reinen" CG-Verfahren im wesentlichen von der Anzahl der Nichtnull-Einträge in M abhängt.

4.2.1 Vorkonditionierung mit ParaSails

Da in unserem Fall A symmetrisch ist, liefert der "setup" von *ParaSails* eine untere Dreiecksmatrix G, die die Inverse der durch Cholesky-Faktorisierung von A entstehenden unteren Dreiecksmatrix L approximiert und das durch die verschiedenen *ParaSails*-Parameter vorher festgelegte Muster ("pattern") besitzt.

Aufgrund der Symmetrie entspricht M^{-1} dem Matrixprodukt $G^T G$, so dass die "Anwendung" von M^{-1} auf r_{i+1} den zusätzlichen Aufwand der Berechnung von $G^T G r_{i+1}$ bedeutet. Hierbei ist nun die Anzahl der Nichtnull-Einträge der Matrix G, $nz(G)$, entscheidend (diese Zahl wird in den Statistik-Angaben zum "setup" von G geführt): je dünner G besetzt ist, desto weniger Mehraufwand wird durch die Vorkonditionierung verursacht und umgekehrt.

Da die Berechnung von $G^T G r_{i+1}$ zwei nacheinander ausgeführten Matrix-Vektor-Multiplikationen entspricht, kommen $2nz(G)$ Multiplikationen und $2 \cdot (nz(G) - n)$ Additionen, also $4nz(G) - 2n$ Operationen, pro Iteration zu den "reinen" CG-Operationen hinzu. Bei der Initialisierung am Anfang muss einmal $G^T G r_0$ berechnet werden, was somit auch noch an Mehraufwand berücksichtigt werden muss.

Addiert man dies alles nun zur obigen Formel für den nicht-vorkonditionierten Fall, so erhält man für das CG-Verfahren mit *ParaSails*-Vorkonditionierung folgende Formel für k Iterationen:

$$\text{Ops}(\textit{ParaSails}) = 2nz(A) + 4nz(G) - 2n + k \cdot (2nz(A) + 4nz(G) + 7n)$$

4.2.2 Vorkonditionierung mit Euclid

Auch für *Euclid* lässt sich eine entsprechende Formel ermitteln. Wie bereits erwähnt, hat die Vorkonditionierungsmatrix M bei *Euclid* die Form $M = LU$, wobei L eine untere und U eine obere Dreiecksmatrix ist, und o.B.d.A. die Diagonale von U nur Einsen enthält. Die "Anwendung" von M^{-1} auf den Vektor r pro Iteration bedeutet das Lösen der beiden Gleichungssysteme

$$(1)\ \ Lw = r \qquad \text{und} \qquad (2)\ \ Us = w$$

Auch hier spielt also die Anzahl der Nichtnull-Einträge in L bzw. U die Hauptrolle bei der Frage nach dem zusätzlichen Rechenaufwand durch das Vorkonditionieren.

Da die Statistik-Angaben zum "setup" von *Euclid* nur die Anzahl der Nichtnull-Einträge der "fill-Matrix" $F = L+U-I$ beinhalten ($nz(F)$), müssen zunächst einmal die zu L und U gehörenden Zahlen $nz(L)$ bzw. $nz(U)$ berechnet werden, die aufgrund der Symmetrie von A identisch sind: $nz(L) = nz(U)$. Weil die Matrix F n Diagonaleinträge besitzt, die alle ungleich Null sind, folgt:

$$nz(L) = nz(U) = (nz(F) - n)/2 + n = \frac{1}{2}nz(F) + \frac{1}{2}n$$

Um nun die Gleichung (1) zu lösen, ist eine Vorwärtssubstitution durchzuführen. Dies bedeutet zunächst $nz(L) - n$ Multiplikationen (die bereits ermittelten Lösungen $1, ..., k$ müssen mit den zugehörigen Koeffizienten in Zeile $k+1$ multipliziert werden, um später die Lösung $k+1$ bestimmen zu können), dann $nz(L) - n$ Subtraktionen (jeweils alle Summanden einer Zeile bis auf die zur Zeile gehörende Unbekannte müssen auf die andere Seite gebracht werden) und n Divisionen (die Werte auf der Diagonalen von L sind größer oder gleich 1, deshalb muss durch den jeweiligen Koeffizienten der "Zeilenunbekannten" geteilt werden, um die Lösung zu bestimmen).

Insgesamt ergibt dies $\frac{1}{2}nz(F) + \frac{1}{2}n$ Multiplikationen und $\frac{1}{2}nz(F) - \frac{1}{2}n$ Additionen, also $nz(F)$ Operationen pro Iteration.

Analog wird nun mit Gleichung (2) verfahren, mit dem Unterschied, dass hier eine Rückwärtssubstitution durchzuführen ist, und die Diagonale von U nur Einsen enthält, so dass die n Divisionen im Vergleich zu (1) wegfallen. Somit ergeben sich für (2) $nz(F)-n$ Operationen und damit insgesamt pro "Anwendung" von M^{-1} $2nz(F)-n$ Operationen, die pro Iteration und auch einmal zu Beginn zur Initialisierung zusätzlich anfallen.

Zusammen mit der "reinen" CG-Formel erhalten wir für das CG-Verfahren mit *Euclid*-Vorkonditionierung schließlich folgende Formel für k Iterationen:

$$\text{Ops}(\textit{Euclid}) = 2nz(A) + 2nz(F) - n + k \cdot (2nz(A) + 2nz(F) + 8n)$$

5 Das 2D-Poisson-Problem

Nachdem wir nun alle theoretischen Grundlagen und Vorbereitungen beisammen haben, können wir uns dem Testen von *Euclid* und *ParaSails* zuwenden.

5.1 Problembeschreibung

Das erste Problem, für dessen Lösung wir *Euclid* und *ParaSails* als Vorkonditionierer des CG-Verfahrens untersucht haben, ist das 2D-Poisson-Problem. Dabei ist folgende Differenzialgleichung zu lösen:

$$\begin{aligned} -\Delta u &= f(x,y)\ , \quad (x,y) \in \Omega \\ u &= g(x,y)\ , \quad (x,y) \in \partial\Omega \end{aligned}$$

wobei Ω in unserem Fall das Gebiet (0,1) x (0,1) ist, f und g Funktionen: $\mathbb{R}^2 \to \mathbb{R}$ sind, und Δ der Laplace-Operator ist mit $\Delta u = \text{div grad } u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = u_{xx} + u_{yy}$.

Die Lösung dieses Dirichletschen Problems (u nimmt auf dem Rand von Ω vorgegebene Werte an)

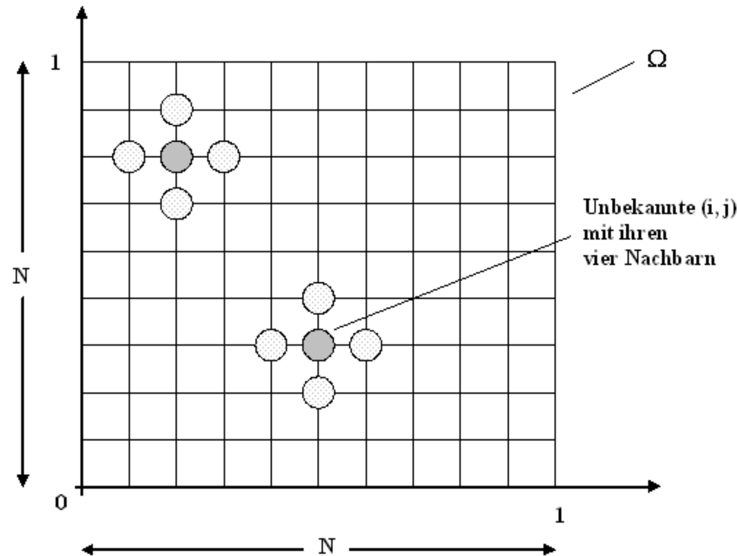


Abbildung 1: Gitternetz beim Poisson-Problem

gibt also das Potenzial des Vektorfeldes $\vec{V} = \text{grad } u$ an.

Um den Computer zur Bearbeitung des Problems einsetzen zu können, mit dem Ziel, eine numerische Approximation von u zu erhalten, muss es zunächst diskretisiert werden, um daraus ein LGS zu erhalten.

Dazu "legen" wir auf das quadratische Gebiet Ω ein regelmäßiges Gitternetz mit einheitlicher Maschenweite h. Jeder Gitterpunkt repräsentiert dabei eine Unbekannte (siehe Abbildung 1). Je kleiner h gewählt wird, desto mehr Unbekannte ergeben sich und desto genauer wird das Ergebnis. Zur Identifizierung der Gitterpunkte (Unbekannten) (i, j) wird eine lexikographische Ordnung festgelegt. Für die Diskretisierung unserer Wahl von Δu ergibt sich – z. B. aus der Betrachtung der Taylorentwicklung von Δu – der folgende Fünf-Punkt-"stencil":

$$[\Delta u]_{ij} \approx \frac{1}{h^2} \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} u$$

Dies bedeutet, dass für $-\Delta u = f(x, y)$ ein LGS der Form $Ax = b$ entsteht, dessen Koeffizientenmatrix A in jeder Zeile maximal 5 Einträge besitzt (für Gitterpunkte, die an die Ränder oder die Ecken von Ω angrenzen, ergeben sich 4 bzw. 3 Einträge). Jede Zeile gehört dabei zu einem bestimmten Gitterpunkt. Diese "Zeilenunbekannte" besitzt den Koeffizienten 4, der immer auf der Hauptdiagonalen von A sitzt, ihre "Nachbarunbekannten" haben jeweils den Koeffizienten -1. Aufgrund des speziellen "five-point-stencil"-Musters ist die dünnbesetzte Matrix A symmetrisch und wegen der Dirichletränder außerdem regulär. Da A zudem positiv-definit ist, ist sie für die Anwendung des (vorkonditionierten) CG-Verfahrens geeignet.

Sei N die Anzahl der Gitterzellen in Ω entlang einer Koordinatenachse (also $h = \frac{1}{N}$). Dann besitzt das Gleichungssystem $(N-1)^2$ Unbekannte, und A ist eine $(N-1)^2$ x $(N-1)^2$ - Matrix.
Die Zahl der Nichtnull-Einträge in A berechnet sich aus:

$$nz(A) = (N-3)^2 \cdot 5 + 4(N-3) \cdot 4 + 4 \cdot 3$$

Für $N = 32$ beispielsweise ergibt sich somit: A ist eine 961 x 961 - Matrix, hat 4380 Nichtnull-Einträge und besitzt folglich eine "sparsity ratio"[8] von 0.47 % (→ dünn-besetzt).

[8] Die "sparsity ratio" bezeichnet den Anteil der Anzahl der Nichtnull-Einträge einer Matrix an der Gesamtanzahl der Einträge.

Die rechte Seite b des Gleichungssystems ergibt sich aus den Funktionswerten von f an den zur jeweiligen Zeilenunbekannten gehörenden Gitterpunkten (multipliziert mit dem Faktor h^2). Für die an die Ränder von Ω grenzenden Punkte muss auch noch die Funktion g mit eingerechnet werden.

In unseren Tests ist die Situation jedoch vereinfacht, da $g = 0$, und f nicht explizit als Funktion betrachtet wird, sondern für die rechte Seite ein vorgegebener Vektor, z. B. $(1, 1, ..., 1)^T$ oder $(1, 0, 0, ..., 0, -1)^T$ gewählt wurde.

Da wir nun sowohl die Koeffizientenmatrix A, als auch den Vektor der rechten Seite b festgelegt haben, das LGS $Ax = b$ also vollständig spezifiziert ist, können wir jetzt das (mit *Euclid* oder *ParaSails* vorkonditionierte) *hypre*-CG-Verfahren zur Lösung des 2D-Poisson-Problems anwenden.

5.2 Testergebnisse

Das Poisson-Problem eignet sich gut, um die grundsätzlichen Eigenschaften von *Euclid* und *ParaSails* und die Auswirkungen verschiedener Parametereinstellungen zu untersuchen. Tabelle 1 zeigt das Ergebnis eines unserer ersten Tests: In Abhängigkeit von verschiedenen Größen des Gitters Ω werden – bei einer rechten Seite von $(1, 0, ..., 0, -1)^T$ – die Iterationszahlen, die das "reine" CG-Verfahren und das mit *Euclid* bzw. *ParaSails* vorkonditionierte Verfahren bis zum Erreichen der Toleranzgrenze von $\epsilon = 10^{-10}$ des relativen Residuums in der 2-Norm (siehe auch Abschnitt 1.1) benötigen, ermittelt. Das *Euclid-level* ist dabei 1, während bei *ParaSails* gilt: *thresh* $= 0.1$, *level* $= 1$, *filter* $= 0\%$.

Beim Poisson-Problem ist der entscheidende Wert für den Schwellwert *thresh* genau 0.25, da alle Nichtnull-Einträge außerhalb der Hauptdiagonalen der Koeffizientenmatrix betragsmäßig immer gleich 1 sind und alle Diagonalelemente den Wert 4 besitzen (siehe Formel für $\tilde{A}$ in Abschnitt 3.1). Deshalb ergibt sich für *thresh* > 0.25 das SPAI-0-Verfahren, für *thresh* < 0.25 dagegen besitzt die Matrix $\tilde{A}$ das "pattern" von A. Statt *thresh* $= 0.1$ könnten wir hier also auch jeden anderen Wert zwischen 0 und 0.25 nehmen, das Ergebnis wäre immer das gleiche.

In Tabelle 1 wird zum einen klar, dass beide Vorkonditionierer die Iterationenanzahl im Vergleich zum nicht-vorkonditionierten Fall verringern, wobei *Euclid* deutlich niedrigere Zahlen aufweist als *ParaSails* – eine Charakteristik, die in unseren sämtlichen weiteren Experimenten bestätigt werden wird. Zum anderen zeigt sich, dass der Nullvektor als Startvektor im Vergleich zu anderen Startvektoren die günstigere Wahl darstellt. Interessant ist außerdem die Tatsache, dass in sämtlichen Fällen eine Verdopplung des Wertes von N auch in etwa eine Verdopplung der benötigten Iterationenanzahl zur Folge hat. Dies bestätigt, dass die Iterationenzahl beim CG-Verfahren von der Komplexität $O(\sqrt{\kappa})$ ist [9] (siehe auch [17]), und zwar sowohl im nicht-vorkonditionierten, als auch im vorkonditionierten Fall. Die Vorkonditionierung verändert (verringert) also nur eine bestimmte Konstante, die grundsätzliche Komplexität des Verfahrens bleibt die gleiche wie vorher.

Als nächstes ist die Frage nach dem Einfluss der verschiedenen levels, sowohl bei *Euclid*, als auch bei *ParaSails*, interessant. Tabelle 2 zeigt die Ergebnisse für *Euclid*, Tabelle 3 die von *ParaSails* (siehe auch Abbildung 2).

Die Anzahl der Gitterpunkte in Ω ist diesmal konstant (N=128), der Startvektor ist jeweils $\vec{0}$, die rechte Seite $\vec{1}$ und ϵ wieder 10^{-10}. Die angegebenen Zeitangaben entsprechen immer der user-time in Sekunden.[10] Der *thresh*-Wert bei *ParaSails* ist wiederum kleiner als 0.25.

Die "nz-ratio" in Tabelle 3 gibt das Verhältnis der Anzahl an Nichtnull-Einträgen der Matrix G (siehe Abschnitt 3.1) des jeweiligen levels zur Matrix G bei level 0 und einem *filter*-Wert von 0% an. Die absoluten nz-Zahlen stehen für die nonzero-Einträge der Matrix $G + G^T - I$ bei einem *filter*-Wert von 0%, entsprechen also in gewisser Weise den Angaben zur Fill-Matrix F (siehe Abschnitt 3.2) bei *Euclid*. Damit lassen sich beide Zahlen miteinander vergleichen, wobei auffällt, dass

[9] In unserem Modellproblem ist $\kappa = h^{-2}$ (κ ist die Konditionszahl der Matrix, h ist die Maschenweite des Gitternetzes). Also gilt: $\sqrt{\kappa} = \frac{1}{h} = N$. Folglich sind die Iterationenzahlen von der Komplexität $O(N)$

[10] Soweit nicht anders angegeben, beziehen sich Zeitmessungen immer auf einen Linux-PC mit AMD Athlon Prozessor, 700 MHz und 768 MB RAM, siehe auch Abschnitt 9.3

N	Startvektor $\vec{0}$			Startvektor zufällig		
	CG	mit Euclid	mit ParaSails	CG	mit Euclid	mit ParaSails
16	36	13	24	83	24	52
32	73	23	43	173	43	101
64	144	42	71	345	80	202
128	274	75	137	684	154	400
256	518	127	260	1346	299	793

Tabelle 1: Euclid (*level* = 1) und ParaSails (*thresh* < 0.25, *level* = 1, *filter* = 0%) bei verschiedenen Gitter-Maschenweiten

level	#it	setup-time	solve-time	total-time	nz von F
0	115	1	2	3	80137
1	79	0	2	2	111889
2	65	1	1	2	143389
3	49	0	2	2	206137
4	41	1	2	3	268381
5	34	0	2	2	330121
6	29	1	2	3	391357
7	26	1	1	2	452089
8	23	0	2	2	512317
9	21	1	1	2	572041
10	19	1	1	2	631261

Tabelle 2: Euclid beim Poisson-Problem, N = 128

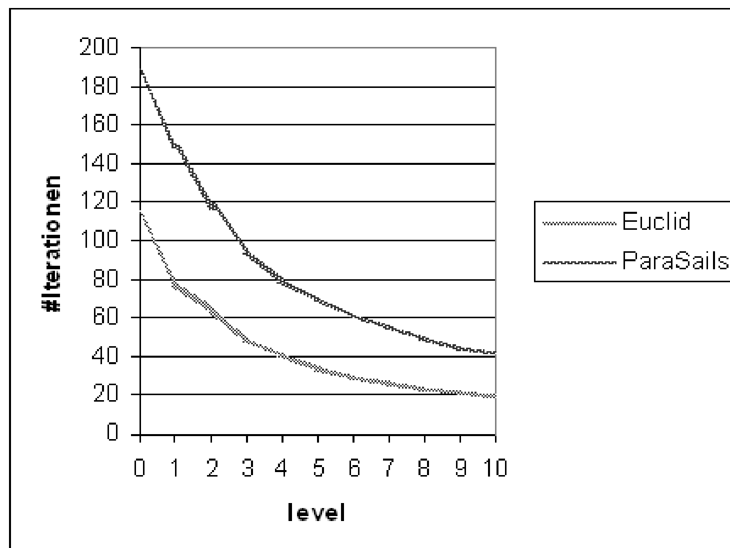


Abbildung 2: Iterationenanzahl bei Euclid und ParaSails (*thresh* < 0.25, *filter* = 0%) abhängig vom *level*

level	#it	setup-time	solve-time	total-time	nz-ratio	nz
filter = 0%						
0	188	0	8	8	1.0	80137
1	149	1	8	9	2.32	191079
2	117	1	8	9	4.28	379760
3	95	2	9	11	6.88	630052
4	80	3	10	13	10.10	940029
5	70	5	11	16	13.93	1308727
6	61	11	13	24	18.37	1736148
7	55	14	13	27	23.39	2219404
8	50	19	14	33	29.00	2759456
9	45	29	15	44	35.18	3354380
10	42	31	15	46	41.93	4004175
filter = 50%						
0	205	0	5	5	1.0	
1	178	1	3	4	1.33	
2	155	1	4	5	2.31	
3	115	1	4	5	3.62	
4	95	2	4	6	5.84	
5	80	3	4	7	7.44	
6	70	5	4	9	9.64	
7	63	8	5	13	12.15	
8	58	11	5	16	14.90	
9	53	25	6	31	18.28	
10	49	27	6	33	21.37	
filter = 90%						
0	205	1	5	6	0.99	
1	282	1	6	7	0.66	
2	226	1	5	6	0.98	
3	199	1	4	5	1.00	
4	182	2	4	6	1.32	
5	163	2	5	7	1.65	
6	168	4	5	9	1.97	
7	151	7	4	11	2.59	
8	121	10	4	14	3.21	
9	106	17	6	23	4.12	
10	105	24	5	29	4.19	

Tabelle 3: ParaSails beim Poisson-Problem (*thresh* < 0.25), N = 128

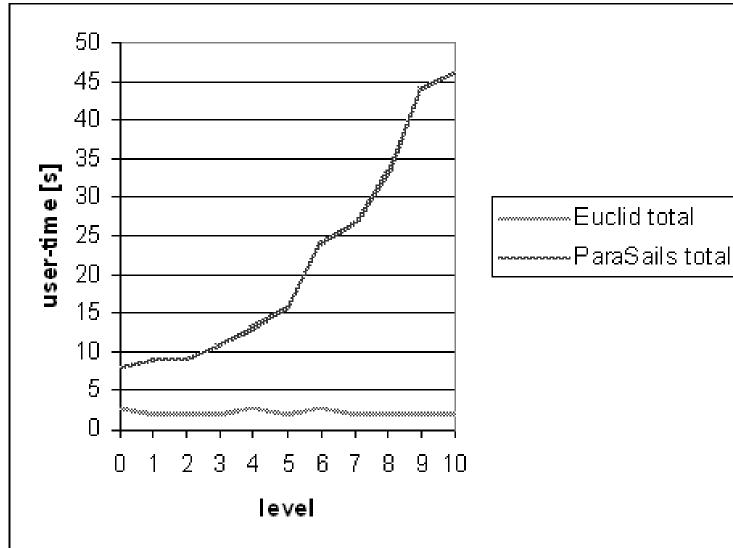


Abbildung 3: Gesamtlösungszeiten beim Poisson-Problem von Euclid und ParaSails
(*thresh* < 0.25, *filter* $= 0\%$), N $= 128$

ParaSails, v. a. in höheren levels, deutlich dichter-besetzte Vorkonditionierungsmatrizen erzeugt als *Euclid*, trotzdem jedoch stets mehr Iterationen benötigt.

Eine weitere Charakteristik von *ParaSails* sind die höheren setup-Kosten, v. a. bei levels größer als 5, die bei *Euclid* einen durchgehend konstant niedrigen Wert annehmen. Die reine Lösezeit ("solve-time") – diese hängt im Prinzip vom "Produkt" der "Dichte" der Vorkonditionierungsmatrix pro level und der Anzahl der insgesamt benötigten Iterationen ab – ist dagegen auch bei *ParaSails* relativ konstant (bis auf den Fall mit *filter* $= 0\%$), jedoch auf deutlich höherem Niveau als bei *Euclid*.

Die Auswirkungen dieser Eigenschaften auf die Gesamtlösezeit "total-time" ("solve-time" + "setup-time") wird u. a. in Abbildung 3 ersichtlich, wo sich die deutliche Überlegenheit von *Euclid* gegenüber *ParaSails* zeigt, während der Effekt der verschiedenen *filter*-Werte bei *ParaSails* auf die Iterationenanzahlen noch einmal anschaulich in Abbildung 4 dargestellt ist: hohe Prozentwerte des *filter* ergeben eine dünner besetzte, ungenauere Matrix und somit mehr Iterationen. In dieser Grafik lässt sich interessanterweise auch gut erkennen, dass zwischen der Iterationenzahl, die sich bei *filter* $= 0\%$ ergibt, und der bei *filter* $= 50\%$ kaum ein Unterschied besteht. Da jedoch die Vorkonditionierungsmatrix beim 50%-*filter*-Wert nur noch etwa die Hälfte ihrer Einträge besitzt wie bei der Situation mit *filter* $= 0\%$ (siehe "nz-ratio" in Tabelle 3), ist der Berechnungsaufwand bei ihrer "Anwendung" natürlich deutlich geringer. Zusammen bedeutet dies eine Einsparung an insgesamter Berechnungszeit, was bei einem Vergleich der Zeitangaben der "solve-time" zwischen *filter* $= 0\%$ und *filter* $= 50\%$ in Tabelle 3 bestätigt wird.

Setzt man übrigens das "reine" CG-Verfahren ohne Vorkonditionierung ein, ergeben sich (für N=128) 264 Iterationen und eine Gesamtlösezeit von 2 s, so dass sich die Vorteile der Vorkonditionierung in diesem Fall noch in Grenzen halten. Grund hierfür ist vor allem die relativ kleine Zahl an Unbekannten (16129), so dass der durch die Vorkonditionierung entstehende "overhead" nicht kompensiert werden kann. Wir werden jedoch natürlich noch einige Beispiele kennenlernen, wo die Vorkonditionierung eine klar ersichtliche Verbesserung bringt.

Deutlicher werden alle bis jetzt erhaltenen Erkenntnisse, was die Charakteristika von *Euclid* und *ParaSails* betrifft, in Tabelle 4 und 5, wo die gleiche Situation wie zuvor für ein feineres Gitter mit N=256 (es ergeben sich also 65025 Unbekannte) dargestellt wird. In Tabelle 5 wird vor allem auch sichtbar, dass für *ParaSails* – was wir auch noch später sehen werden – die (zeitlich) optimale Konfiguration meist bei niedrigen *thresh*-, (prozentualen) *filter*- und *level*-Werten erreicht wird,

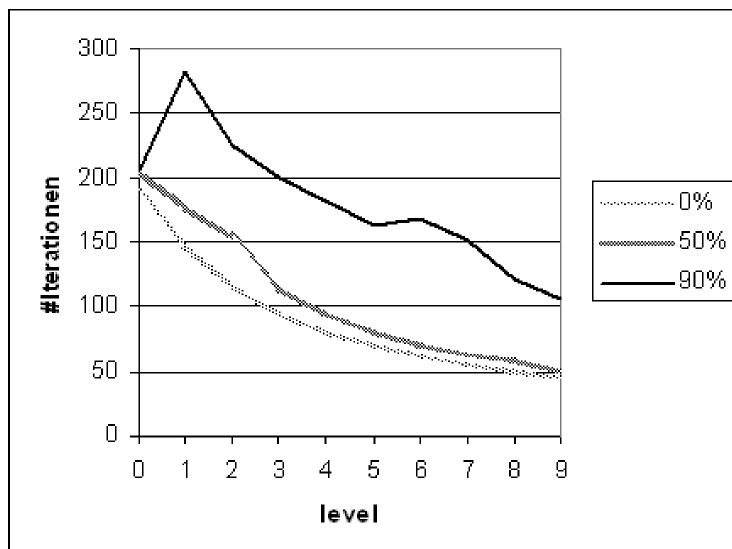


Abbildung 4: Abhängigkeit der Iterationen vom filter-Wert bei ParaSails (*thresh* < 0.25), N = 128

level	#it	setup	solve	total
0	215	2	21	23
1	151	2	16	18
3	91	2	13	15
5	60	3	11	14
9	37	4	11	15

Tabelle 4: Euclid beim Poisson-Problem, N = 256

während *Euclid* in unserem Beispiel bei höheren levels (größer als 5) die beste Performance zeigt (Tabelle 4).

Setzt man im Falle von N=256 den *thresh*-Wert bei ParaSails auf Werte größer als 0.25, so dass das SPAI-0-Verfahren entsteht und deshalb die Anzahl der levels egal ist, weil mit jeder Matrixpotenzierung immer das Diagonalmuster erhalten bleibt, ergeben sich 532 Iterationen bei einer setup-Zeit von 1 s und einer Lösezeit von 44 s. Man sieht also, dass auch dieser Fall, was die Gesamtlösezeit betrifft, bei weitem nicht an die (bei uns) beste Konfiguration mit *thresh* < 0.25, *level* $= 0$ und *filter* $= 0\%$ herankommt.

level	#it	setup-time	solve-time	total-time
filter = 0%				
0	349	1	33	34
1	305	2	33	35
3	185	4	37	41
5	138	11	43	54
7	109	28	53	81
9	90	59	63	122
filter = 50%				
0	386	2	36	38
1	344	2	32	34
3	206	5	30	35
5	145	12	30	42
7	118	31	34	65
9	101	79	43	122
filter = 90%				
0	386	1	43	44
1	533	2	54	56
3	392	5	41	46
5	310	14	40	54
7	283	30	38	68
9	206	73	34	107

Tabelle 5: ParaSails beim Poisson-Problem (*thresh* < 0.25), N = 256

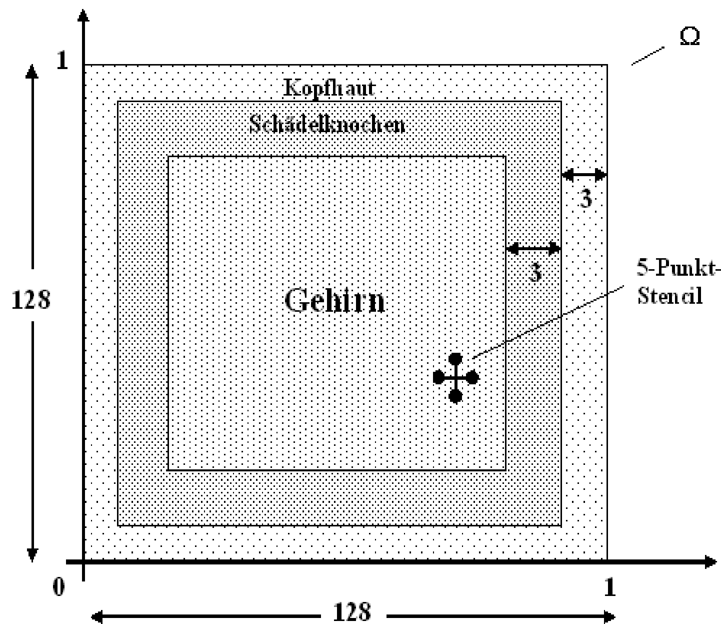


Abbildung 5: Die verschiedenen Bereiche beim verallgemeinerten Poisson-Problem

6 Das verallgemeinerte 2D-Poisson-Problem

6.1 Problembeschreibung

Um auf das realistische "Kopfproblem" hinzuführen, wurde im nächsten Schritt das 2D-Poisson-Problem in der Hinsicht verändert und erweitert, dass nun das quadratische Gebiet Ω verschiedene Bereiche aufweist, denen jeweils ein bestimmter σ-Wert zugeordnet ist ($\rightarrow$ "springende Koeffizienten"). Dieses σ soll die Leitfähigkeit des (in der Realität) im jeweiligen Bereich vorhandenen Gewebes simulieren. Ein hoher σ-Wert bedeutet dabei eine hohe Leitfähigkeit (einen geringen Widerstand), ein kleiner Wert einen großen Widerstand. Zur Vereinfachung unterscheiden wir im Kopf nur drei Gewebearten: Kopfhaut, Schädelknochen und Gehirnmasse, wobei Kopfhaut und Gehirnmasse jeweils die gleiche Leitfähigkeit σ besitzen sollen. Die den Kopf umgebende Luft erhält den σ-Wert 0.
Mathematisch formuliert, ist die folgende Differenzialgleichung zu lösen:

$$\begin{aligned} -\nabla \cdot (\sigma \cdot \nabla u) &= f(x,y)\ , \quad (x,y) \in \Omega \\ \frac{\partial u(x,y)}{\partial n} &= 0\ , \quad (x,y) \in \partial\Omega \end{aligned}$$

Hierbei ist ∇ der Nablaoperator ($\nabla u = \text{grad } u,\ \nabla \cdot \vec{V} = \text{div } \vec{V}$), f eine Funktion: $\mathbb{R}^2 \rightarrow \mathbb{R}$, und Ω wieder das Gebiet (0,1) x (0,1).

Im Gegensatz zum einfachen 2D-Poisson-Problem, haben wir es hier also mit einem Neumannschen Problem zu tun. Zur Diskretisierung verwendet man einen Finite-Differenzen-Ansatz bzw. eine "Box Integration", siehe auch z. B. [15].

Betrachten wir dazu das Gitternetz im Gebiet Ω. Die Bereiche "Kopfhaut" und "Schädelknochen" haben dabei in unseren Tests jeweils die Breite von drei Maschen, das restliche Gebiet soll das Gehirn repräsentieren (siehe Abbildung 5). N hat bei uns den Wert 128 (d. h. $h = \frac{1}{128}$), also haben wir 16129 (127^2) Unbekannte.

Jeder Gitterpunkt hat vier Nachbarn (siehe Abbildung 6), die unter Umständen in einem anderen Gebiet (mit anderem σ-Wert) liegen als der zentrale Punkt.

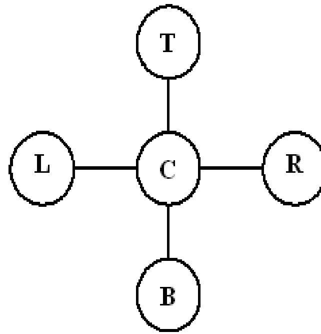


Abbildung 6: 5-Punkt-Stencil beim verallgemeinerten Poisson-Problem

Die jeweiligen Koeffizienten berechnen sich nach der folgenden Formel (als Beispiel sei hier der Koeffizient für den linken Nachbarn K_{L} aufgeführt; die anderen berechnen sich analog):

$$K_{\mathrm{L}} = -2 \cdot \frac{\sigma_{\mathrm{L}} \cdot \sigma_{\mathrm{C}}}{\sigma_{\mathrm{L}} + \sigma_{\mathrm{C}}} \ ,$$

wobei σ_{L} und σ_{C} die im Gebiet von Punkt L bzw. Punkt C geltenden σ-Werte sind.

Der Koeffizient der zentralen Unbekannten ergibt sich als:

$$K_{\mathrm{C}} = -(K_{\mathrm{L}} + K_{\mathrm{B}} + K_{\mathrm{R}} + K_{\mathrm{T}})$$

Man erkennt, dass sich für den Fall, dass alle σ-Werte gleich 1 sind, genau wieder das ursprüngliche einfache 2D-Poisson-Problem ergibt. In unserem Fall gelten die folgenden Werte für σ: $\sigma_{\mathrm{Luft}} = 0$, $\sigma_{\mathrm{Knochen}} = 1$, $\sigma_{\mathrm{Skalp}} = \sigma_{\mathrm{Hirn}} = 16$ (auch 80 und 1000).

Da der Koeffizient, der einen bestimmten Gitterpunkt l mit einem Gitterpunkt k verbindet, identisch ist mit dem Koeffizienten, der k mit l verbindet, ist die resultierende dünn-besetzte Koeffizientenmatrix A des Gleichungssystems symmetrisch. Außerdem sind alle Diagonaleinträge positiv und alle sonstigen Einträge nicht-positiv. Allerdings ist A aufgrund der Neumannschen Ränder singulär, da die Summe der Koeffizienten in jeder ihrer Zeilen Null ergibt ($\rightarrow$ Kern$(A) = (1, 1, ..., 1)^T$). Dies kann aber leicht behoben werden, indem man einfach eine bestimmte Unbekannte auf 0 setzt (z. B. bei uns die zum Gitterpunkt der rechten unteren Ecke von Ω gehörende Unbekannte), und dann die zu dieser Unbekannten gehörende Zeile und Spalte aus der Koeffizientenmatrix streicht (bezüglich des ausgewählten auf 0 gesetzten Punktes werden Dirichlet-Ränder angenommen). Natürlich muss diese Position auch aus dem Vektor der rechten Seite b gestrichen werden. Damit entstehen einige Zeilen in A, deren Summe ungleich Null ist, und die Matrix ist wieder regulär wie beim einfachen 2D-Poisson-Problem.

Da der Vektor der rechten Seite bei uns $(1, 0, 0, ..., 0, -1)^T$ ist, womit ähnlich wie beim späteren realistischen "Kopfproblem" eine Strom-Quelle und eine Senke simuliert wird, und bei symmetrischen Matrizen gilt: Kern$(A) \perp$ Im(A), und das Skalarprodukt aus $(1, 1, ..., 1)^T$ (dem Kern der singulären Matrix A) und $(1, 0, 0, ..., 0, -1)^T$ gleich Null ist, ist $(1, 0, 0, ..., 0, -1)^T$ im Bild der Matrix A enthalten. Daher hat auch das singuläre Problem bei uns eine Lösung (bzw. unendlich viele, die sich jeweils durch eine additive Konstante unterscheiden) und es lässt sich auch gut mit dem mit *Euclid* oder *ParaSails* vorkonditionierten CG-Verfahren lösen. Wir beschränken uns deshalb bei den nun folgenden Tests auf den singulären Fall. Bei den Problemlösungen der weiteren Kapitel werden wir dann jedoch auch einmal den regularisierten Fall mit einbeziehen.

level	#it	setup	solve	total	#it	setup	solve	total
	ParaSails				Euclid			
$\sigma_{\text{Hirn/Skalp}} = 16$								
0	266	0	6	6	148	0	3	3
1	298	1	7	8	131	0	3	3
5	138	3	6	9	48	1	2	3
9	90	15	9	24	29	1	2	3
$\sigma_{\text{Hirn/Skalp}} = 80$								
0	478	0	9	9	179	0	4	4
1	327	1	7	8	158	0	4	4
5	155	2	7	9	54	0	3	3
9	105	12	8	20	31	1	2	3
$\sigma_{\text{Hirn/Skalp}} = 1000$								
0	501	0	10	10	186	1	3	4
1	427	1	9	10	197	0	5	5
5	191	3	8	11	62	1	2	3
9	126	13	9	22	33	1	2	3

Tabelle 6: Euclid und ParaSails (*thresh* < 0.2, *filter* $= 50\%$) beim verallgemeinerten Poisson-Problem, keine Regularisierung, Startvektor $\vec{0}$, $\epsilon = 10^{-10}$

6.2 Testergebnisse

Bei der Lösung des verallgemeinerten Poisson-Problems ergeben sich grundsätzlich die gleichen Verhältnisse zwischen *ParaSails* und *Euclid* wie beim einfachen 2D-Poisson-Problem, d. h. *ParaSails* schneidet wieder deutlich schlechter ab – sowohl bei den Iterationszahlen, als auch bei den setup- und Lösezeiten in allen levels. Allgemein sind die Iterationenzahlen aufgrund des Mehraufwandes durch die springenden Koeffizienten jedoch höher als beim einfachen Poisson-Problem (siehe Tabelle 6).

Bei unseren Tests wurden drei verschiedene Fälle hinsichtlich des Wertes des Leitfähigkeitskoeffizienten des Gehirns/der Haut untersucht: $\sigma_{\text{Hirn/Skalp}} = 16$, 80 und 1000. Dabei fällt auf, dass für größere σ-Werte der Berechnungsaufwand wächst, was sich in steigenden Iterationszahlen niederschlägt (siehe Abbildung 7).

Interessant ist außerdem die Tatsache, dass sich beim verallgemeinerten Poisson-Problem erstmals auch eine zeitliche Verbesserung durch die Vorkonditionierung ergibt. Betrachten wir dazu einmal die Ergebnisse des Einsatzes des nicht-vorkonditionierten CG-Verfahrens zur Lösung des Problems:

$\sigma_{\text{Hirn/Skalp}}$	16	80	1000
#it	468	556	946
solve-time	4	4	9

Vergleicht man diese Daten mit den Ergebnissen aus Tabelle 6, so zeigt sich beispielsweise für $\sigma_{\text{Hirn/Skalp}} = 1000$, dass *Euclid* in allen levels geringere Gesamtlösezeiten aufweist als das "reine" CG-Verfahren: 3 - 5 s vs. 9 s. Dies bedeutet, dass hier durch die Vorkonditionierung nicht nur die Iterationenanzahl verringert wird, sondern insgesamt wirklich weniger elementare Rechenoperationen als beim nicht-vorkonditionierten Fall benötigt werden, um die Toleranzgrenze der Lösung zu erreichen.

ParaSails hingegen kann wieder lediglich bei niedrigen levels (0 oder 1) allenfalls mit der CG-

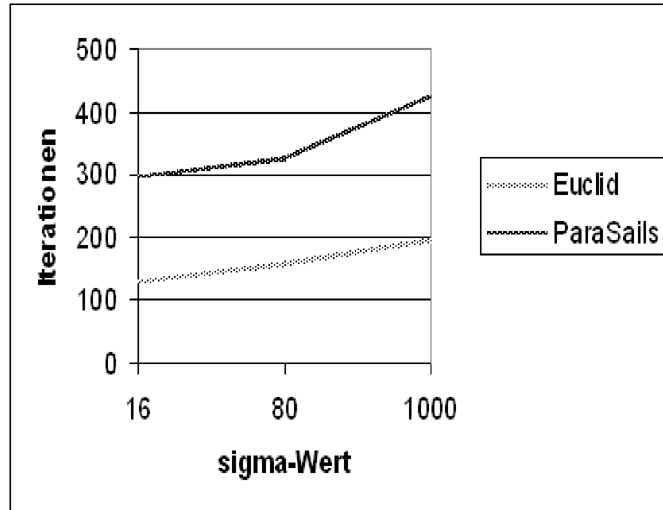


Abbildung 7: Iterationen in Abhängigkeit vom sigma-Wert für Hirn/Skalp beim verallgemeinerten 2D-Poisson-Problem für Euclid (*level* = 1) und ParaSails (*thresh* < 0,2, *level* = 1, *filter* = 50%), keine Regularisierung

Lösezeit gleichziehen, eine wirkliche Verbesserung ist jedoch nicht sichtbar. Bei hohen levels (z. B. 9) zeigen sich zudem erneut die unverhältnismäßig hohen Setup-Kosten.
Es bleibt noch zu bemerken, dass sich bei *ParaSails* der "kritische Wert" für *thresh*, der ja beim einfachen 2D-Poisson-Problem exakt bei 0.25 gelegen hat (siehe auch Abschnitt 5.2), beim jetzigen Problem aufgrund der springenden Koeffizienten auf einen "kritischen Bereich" in etwa zwischen 0.2 und 0.4 ausdehnt (der genaue Bereich hängt natürlich auch vom gewählten Wert für $\sigma_{\text{Hirn/Skalp}}$ ab), so dass *thresh*-Werte kleiner als 0.2 (so wie bei uns mit *thresh* = 0.1) für die Matrix $\tilde{A}$ auf jeden Fall das "pattern" von A liefern (wie bereits erwähnt ergibt sich bei level 0 das SPAI-1-Verfahren), während bei Werten größer als 0.4 die Vorkonditionierungsmatrix nur ein Diagonalmuster (→ SPAI-0) besitzt, und deshalb auch die levels keine Rolle spielen.

7 Das 2D-Slice-Problem

7.1 Problembeschreibung

Das zuletzt untersuchte verallgemeinerte 2D-Poisson-Problem war durch die quadratische Form des Gebietes Ω noch recht unrealistisch. Deshalb betrachten wir jetzt eine auf einem wirklichen Querschnitt eines menschlichen Kopfes basierende Fläche (siehe Abbildung 8). Dies kann als direkte Vorstufe zum dreidimensionalen "Kopfproblem" (Abschnitt 8) gesehen werden.

Die hier zu lösende Differenzialgleichung und die daraus resultierende Diskretisierung sind die gleichen wie beim verallgemeinerten 2D-Poisson-Problem (siehe Abschnitt 6.1), und es gibt auch die gleichen Gewebearten mit den dazugehörigen Werten $\sigma_{\text{Luft}} = 0$, $\sigma_{\text{Skalp}} = 16 = \sigma_{\text{Hirn}}$ und $\sigma_{\text{Knochen}} = 1$. Natürlich sind die verschiedenen Gewebearten nun nach den realistischen Gegebenheiten im Gebiet des Kopfquerschnitts angeordnet. Zusätzlich werden allerdings jetzt noch die sog. *Ventrikel* mit einbezogen – ein Bereich in der Mitte des Gehirns, der mit einer Flüssigkeit, dem sog. *Liquor*, gefüllt ist (siehe Abbildung 8), dessen Leitfähigkeitskoeffizient $\sigma_{\text{Ventrikel}}$ den Wert 128 besitzt. Der Liquor leitet demnach Strom deutlich besser als das normale Hirngewebe oder die Haut. In unseren Tests wird die Situation mit und ohne Ventrikeln betrachtet.

Gegeben ist der Datensatz des Kopfquerschnitts in Form einer Binärdatei, die für jeden Punkt eines 512 x 512 Einheiten großen quadratischen Gitternetzes (es gibt dort also 262144 Punkte) einen "Marker" gespeichert hat, der für eine der insgesamt vier (bzw. fünf, mit Ventrikeln)

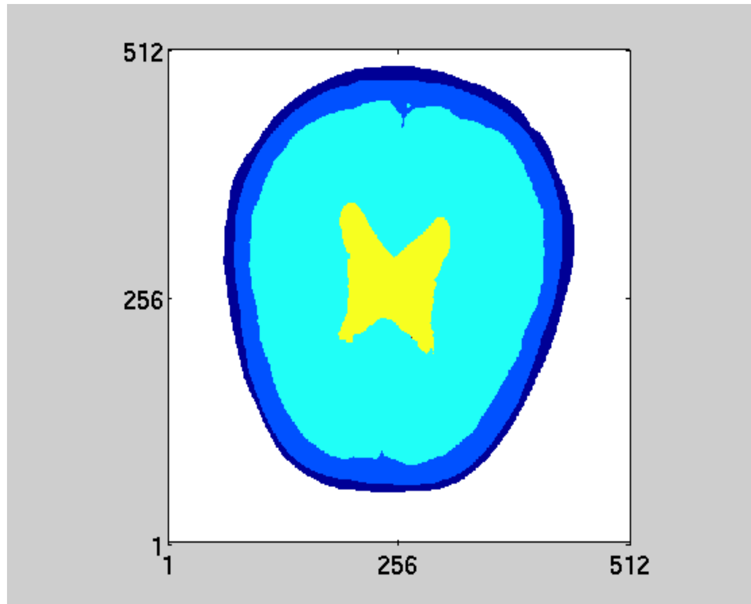


Abbildung 8: Gehirn-Querschnitt beim Slice-Problem mit den verschiedenen Geweberegionen Skalp, Knochen, Hirn (von außen nach innen) und dem Bereich der Ventrikel mit dem Liquor in der Mitte

Gewebetypen (inklusive Luft) steht. Da nur der eigentliche Kopfquerschnitt, der lediglich etwa die Hälfte der quadratischen Fläche ausfüllt, betrachtet werden muss, und nicht auch die Luft, müssen zunächst diejenigen Punkte ermittelt werden, die Teil des Querschnitts sind. In unserem Fall ergeben sich 136443 solcher Punkte, die dann die Unbekannten des aus der Diskretisierung resultierenden Gleichungssystems darstellen. In einem weiteren Durchlauf dieser Punkte werden anschließend die σ-Werte der jeweiligen zentralen "Zeilen-Unbekannten" und deren vier Nachbarn ($\rightarrow$ "five-point-stencil") festgestellt und aus den daraus (mit der gleichen Formel wie beim verallgemeinerten 2D-Poisson-Problem) berechneten Koeffizienten das LGS aufgestellt.

Auch hier entsteht eine symmetrische, dünn-besetzte Koeffizientenmatrix A mit positiven Diagonalelementen und nicht-positiven Außer-Diagonalelementen, die aufgrund der Neumannschen Ränder allerdings ebenfalls zunächst singulär ist, jedoch nach dem gleichen Prinzip wie beim verallgemeinerten 2D-Poisson-Problem (siehe Abschnitt 6.1) regularisiert werden kann.

Der Vektor der rechten Seite b wird wiederum so gewählt, dass er an einer bestimmten Stelle – bei uns im Punkt $E_1(250|54)$[11] – den Wert 1 (simuliert Strom-Quelle), an einer anderen Stelle – bei uns in $E_2(167|476)$ – den Wert -1 (simuliert Strom-Senke), und sonst überall den Wert 0 besitzt, so dass sich bei der Regularisierung beispielsweise der genau in der Mitte des Kopfquerschnitts liegende Punkt (256|256) als die von vornherein auf Null zu setzende Unbekannte anbietet. In unseren Tests betrachten wir sowohl den singulären, als auch den regularisierten Fall.

7.2 Testergebnisse

Bei unseren Messungen für das Slice-Problem wurde sowohl der Fall mit Ventrikeln, als auch ohne betrachtet. Die Tabellen 7 und 8 zeigen die Ergebnisse für *Euclid* und *ParaSails* für die nicht-regularisierte Koeffizientenmatrix, wobei bei ParaSails ein *thresh*-Wert von 0.1 gewählt wurde, was auch hier deutlich unter dem "kritischen Bereich" zwischen 0.2 und 0.3 liegt ($\rightarrow \tilde{A}$ hat Muster von A). Die Toleranzgrenze ϵ beträgt 10^{-6}.

Im Vergleich zu den beiden vorangegangenen Problemen zeigt sich, dass die Iterationenanzahlen bei beiden Vorkonditionierungsverfahren nun deutlich höher liegen, was natürlich zum einen

[11] Die Koordinaten beziehen sich auf das 512 x 512 - Gitter, siehe auch Abbildung 8.

level	mit Ventrikeln				ohne Ventrikeln			
	#it	setup	solve	total	#it	setup	solve	total
0	480	3	98	101	491	4	89	93
3	161	4	44	48	162	4	45	49
7	70	7	32	39	71	7	33	40
9	53	8	29	37	54	9	28	37

Tabelle 7: Euclid beim Slice-Problem, keine Regularisierung, Startvektor $\vec{0}$

level	mit Ventrikeln				ohne Ventrikeln			
filter = 50%								
	#it	setup	solve	total	#it	setup	solve	total
0	872	2	163	165	880	2	165	167
3	434	10	127	137	448	10	128	138
5	305	25	125	150	313	27	126	153
7	240	65	137	202	250	69	142	211
filter = 90%								
0	1499	2	252	254	1653	2	256	258
3	801	9	153	162	892	9	155	164
5	573	23	130	153	640	26	142	168
7	493	60	120	180	537	63	126	189

Tabelle 8: ParaSails beim Slice-Problem ($thresh < 0.2$), keine Regularisierung, Startvektor $\vec{0}$

in der klar größeren Anzahl an Unbekannten (immerhin 136443 vs. 16429), zum anderen in der unstrukturierteren und unregelmäßigeren Verteilung der einzelnen Geweberegionen im realistischen Kopfquerschnitt begründet ist. Was den Vergleich zwischen *Euclid* und *ParaSails* betrifft, wird auch hier wieder die klare Überlegenheit von *Euclid* deutlich: Der damit erzielten optimalen Gesamtlösezeit von 37 s (die interessanterweise erst bei *level* 9 erreicht wird; vorher sinkt die benötigte Zeit mit jedem höheren *level*, siehe Tabelle 7) steht bei *ParaSails* eine entsprechende Zeit von 137 s (bei *level* 3, *filter* = 50%) gegenüber. Die wiederum sehr hohen setup-Kosten von *ParaSails* ab *level* 5 müssen sicherlich gesondert berücksichtigt werden, betrachtet man aber die reinen Lösezeiten (solve-time), so ist auch dort der Unterschied zu *Euclid* gravierend. Dies bedeutet, dass bei *ParaSails* trotz deutlich dichter besetzter und damit eigentlich genaueren Vorkonditionierungsmatrizen die daraus resultierenden Iterationenanzahlen immer noch unverhältnismäßig hoch sind, wodurch natürlich insgesamt weitaus mehr elementare Operationen bis zum Erreichen der "Lösungsschwelle" ausgeführt werden müssen als bei *Euclid*.

Übrigens beträgt die Gesamtlösezeit für den in Tabelle 8 nicht berücksichtigten Fall bei *ParaSails*, dass *filter* = 0% und gleichzeitig *level* = 0 (→ SPAI-1), 166 s (bei 2 s setup-Zeit und 830 Iterationen), so dass auch hier keine bessere Performance erkennbar ist.

Betrachten wir nun einmal das nicht-vorkonditionierte CG-Verfahren für die Lösung des Problems. Es ergeben sich dabei folgende Werte:

Ventrikel	solve-time	#Iterationen
ja	483	5184
nein	179	2020

Während also beim "reinen" CG-Verfahren das Fehlen der Ventrikel eine klare Verringerung des Berechnungsaufwandes – resultierend in einer mehr als halbierten Iterationenanzahl und Rechenzeit gegenüber der Situation mit den Ventrikeln – mit sich bringt, ist beim vorkonditionierten Fall – sowohl bei *Euclid*, als auch bei *ParaSails* – kaum ein Unterschied zu erkennen, und wenn überhaupt, dann in genau umgekehrter Weise als oben (siehe Tabellen 7 und 8). Dies bedeutet, dass beim "reinen" CG-Verfahren die Hinzunahme der Ventrikel einen deutlich negativeren Einfluss auf die Konditionszahl der Koeffizientenmatrix hat, als im vorkonditionierten Fall.

Wichtiger ist hier jedoch sicherlich die Erkenntnis, dass beim Slice-Problem nun wirklich deutlich die Vorteile des Vorkonditionierens erkennbar werden: Einer bestmöglichen Gesamtlösezeit von 483 s beim "reinen" CG-Verfahren für den Fall mit Ventrikeln stehen 37 s bei *Euclid* und 137 s bei *ParaSails* gegenüber. Selbst *ParaSails* kann hier also erstmals klare Verbesserungen bezüglich der Lösezeit liefern.

Für den regularisierten Fall ergeben sich im Grunde genau die gleichen Verhältnisse, mit dem einzigen Unterschied, dass die Iterationenanzahlen überall um ca. 15-20% erhöht sind. Als Beispiel dafür seien hier noch kurz die Daten für das nicht-vorkonditionierte CG-Verfahren angeführt:

Ventrikel	solve-time	#Iterationen
ja	547	6018
nein	228	2522

8 Das 3D-Kopfproblem

8.1 Problembeschreibung

Wie bereits in der Einführung angedeutet, ist die Aufgabe der Rekonstruktion der elektrischen Hirnaktivität bzw. der Lokalisierung von Stromquellen im Gehirn anhand von EEG-Daten – man spricht hier auch von einem *inversen* Simulationsproblem – die zentrale Motivation dieser Studienarbeit [16]. Die *hypre*-Vorkonditionierer *Euclid* und *ParaSails* sollen dabei zur Lösung mehrerer sog. Vorwärtsprobleme, die ein wesentlicher Bestandteil der Lösung des inversen Problems sind, eingesetzt und getestet werden.

Der Hauptaufwand bei der Lösung des inversen EEG-Problems besteht im Aufstellen der sog. "lead field matrix" für einen bestimmten Punkt des Gehirns. Diese Matrix wird so konstruiert, dass sie die "Auswirkungen" eines an diesem Punkt existierenden und in eine gewisse Richtung orientierten Dipols (Modellvorstellung) auf die vorhandene (begrenzte) Anzahl n von EEG-Elektroden auf der Kopfhaut abbildet [16]. Vermutet man nun in diesem Punkt eine elektrisch aktive Hirnregion (beispielsweise bei einem Epilepsie-Patienten), modelliert als "Dipol-Vektor" mit einer bestimmten Position und Richtung, kann man durch eine einfache Matrix-Vektor-Multiplikation zwischen dem "Richtungsvektor" des Dipols und der zu der besagten Position gehörenden "lead field matrix" die sich ergebenden Potenzialwerte an den Elektroden bestimmen und dann mit den tatsächlich gemessenen vergleichen. Auf diese Weise würde ein Optimierungsproblem entstehen, mit dem Ziel, die passendste Dipolanordnung zu finden.

Aufgrund des Reziprozitäts-Theorems von Helmholtz, lässt dich die "lead field matrix" folgendermaßen aufstellen [16]:

Für $(n-1)$ repräsentative Elektrodenpaare (A, B) wird die Potenzialverteilung im Kopf berechnet, wenn ein Strom I_{AB} bei Elektrode A in den Kopf eintritt (entspricht Quelle) und in Elektrode B wieder austritt (Senke), oder umgekehrt. Diese Berechnungsaufgabe wird als *Vorwärtsproblem* bezeichnet. In unserem Fall haben wir insgesamt 27 Elektroden, so dass 26 solcher Vorwärtsprobleme zu lösen sind.

Mathematisch ausgedrückt, ist die folgende Differenzialgleichung für jedes Elektrodenpaar zu lösen (3D-Poisson-Problem) [16]:

$$\begin{aligned} \nabla \cdot (\sigma \cdot \nabla \Phi(x)) &= -I\delta(x_{\text{Quelle}} - x) + I\delta(x_{\text{Senke}} - x) \ , \quad x \in \Omega \\ \sigma \frac{\partial \Phi(x)}{\partial n} &= 0 \ , \quad x \in \partial\Omega \ , \end{aligned}$$

wobei $x \in \mathbb{R}^3$, δ die Diracsche delta-Funktion, und $\Phi(x)$ die gesuchte Potenzialverteilung (mit $\Phi(x) : \mathbb{R}^3 \to \mathbb{R}$) ist, und I den Strom darstellt, der zwischen den zwei Elektroden fließt.

Ist Φ berechnet, kann man die "lead field matrix" für eine beliebige Anzahl an Dipolen, oder für ein Dipol an einer beliebigen Anzahl von Positionen, durch einen einfachen Interpolationsprozess aufstellen [16].

Trotzdem bleibt die Lösung von $n-1$ solcher Vorwärtsprobleme an einem realistischen Kopfmodell, v. a. wenn n groß ist, ein kostspieliges Unterfangen, weshalb effektive und effiziente Lösungsverfahren ($\rightarrow$ Vorkonditionierer, *ParaSails* und *Euclid*) eine wichtige Voraussetzung sind.

Um den Computer zur Lösung der obigen Differenzialgleichung einsetzen zu können, muss das Problem wie immer diskretisiert werden. Dazu wird der Kopf in lauter sehr kleine Würfel, sog. Voxel, eingeteilt, wobei jedes Voxel den zu seinem Ort gehörenden Wert der zu berechnenden Potenzialverteilung Φ repräsentiert und somit in dem zu erstellenden LGS jeweils eine Unbekannte darstellt.

Genauso wie beim verallgemeinerten 2D-Poisson- oder 2D-Slice-Problem (siehe Abschnitte 6 und 7) werden auch hier die drei verschiedenen Gewebearten Skalp, Knochen und Gehirn (keine Ventrikel) mit den dazugehörigen Leitfähigkeitskoeffizienten σ mit einbezogen (bei uns: $\sigma_{\text{Skalp}} = \sigma_{\text{Hirn}} = 16, \ \ \sigma_{\text{Knochen}} = 1, \ \ \sigma_{\text{Luft}} = 0$), wobei die Annahme, dass die Gewebetypen konstante und isotrope Leitungseigenschaften besitzen, natürlich eine starke Vereinfachung dar-

stellt. Mit der heutigen modernen Technologie in der Medizin (CT[12] und v. a. MRI[13]) ist es möglich, die wirkliche Geometrie des Kopfes eines individuellen Patienten für die Festlegung des Problembereichs Ω zu verwenden, sowie die genaue Verteilung der verschiedenen Gewebearten im Kopf zu ermitteln und als Datensatz zur Verfügung zu stellen. Auch wir verwenden einen solchen realistischen Datensatz.

Benützt man einen Finite-Differenzen-Ansatz mit "Box-Integration", siehe auch z. B. [15], so ergibt sich, dass für jedes Voxel V_l, das zum Kopf gehört, die diskrete Approximierung Φ_h der obigen Differenzialgleichung der folgenden Gleichung genügen muss [16]:

$$\left(\sum_{k\in N_l} \gamma_k\right) \Phi_l \quad - \quad \sum_{k\in N_l} \gamma_k \Phi_k \quad = \quad I(\delta_{k_{\text{Quelle}},l} \quad - \quad \delta_{k_{\text{Senke}},l}) \ ,$$

wobei N_l die Menge der Indizes der sechs Nachbarn von Voxel V_l ist ($\rightarrow$ dreidimensionaler "Sieben-Punkt-Stencil"), $\delta_{i,j}$ das Kronecker-Symbol bezeichnet, und die Koeffizienten γ_k sich ebenso wie beim verallgemeinerten 2D-Poisson- oder 2D-Slice-Problem aus den in Voxel V_l bzw. dessen direkter Umgebung (Nachbarn) vorliegenden σ-Werten berechnen:

$$\text{z. B.} \quad \gamma_{\text{Ost}} \quad = \quad 2 \cdot \frac{\sigma_l \cdot \sigma_{\text{Ost}}}{\sigma_l + \sigma_{\text{Ost}}} \ , \qquad \text{analog für die anderen fünf Nachbarkoeffizienten}$$

In unserem Fall ist das Kopfmodell in 70830 Voxel eingeteilt, das resultierende LGS hat also genausoviele Unbekannte.

Die dazugehörige Koeffizientenmatrix A ist dünn-besetzt (482184 Nichtnull-Einträge $\rightarrow$ "sparsity ratio": $\approx 0.01\%$), symmetrisch, (schwach) diagonal-dominant, sie besitzt positive Diagonal- und nicht-positive Außer-Diagonaleinträge und ist – wie die Matrizen beim verallgemeinerten 2D-Poisson- und 2D-Slice-Problem auch – singulär, da in jeder Zeile die Summe aller Einträge gleich Null ist, so dass $\text{Kern}(A) = (1, 1, ..., 1)^T$. Auch hier besteht wieder die Möglichkeit, A nach dem bereits erwähnten Prinzip zu regularisieren. Wir betrachten bei unseren Tests beide Fälle.

Die rechte Seite b eines solchen aus dem Vorwärtsproblem entstandenen Gleichungssystems hat immer die Form $(0, 0, ..., 0, 1, 0, ..., 0, -1, 0, ..., 0)^T$, wobei die Positionen der "1" und der "-1" im Vektor vom Ort der beiden relevanten Elektroden des jeweiligen Vorwärtsproblems abhängen. Somit besitzt das Gleichungssystem auch im singulären Fall eine Lösung (bzw. unendlich viele, siehe auch Abschnitt 6.1). Auf das Ergebnis der letztendlichen Lösung des inversen Kopfproblems hat die additive Konstante, durch die sich bei einer singulären Koeffizientenmatrix die Lösungen eines Vorwärtsproblems voneineinder unterscheiden können, keinen Einfluss.

8.2 Parameterstudie

Um unsere 26 Vorwärtsprobleme mithilfe von *Euclid* und *ParaSails* möglichst effizient zu lösen, ist zunächst einmal eine Parameterstudie nötig, um für beide Verfahren die optimale Parameterkonfiguration hinsichtlich der Lösezeit (bzw. der insgesamt benötigten elementaren arithmetischen Operationen) zu finden. Dazu haben wir zwei bestimmte Elektrodenpaare betrachtet und die beiden dazugehörigen Vorwärtsprobleme gelöst. Die Toleranzgrenze ϵ für das relative Residuum in der 2-Norm wurde hierbei auf 10^{-8} gesetzt.

Die setup-Kosten für die Vorkonditionierer bleiben hier unberücksichtigt, da die Vorkonditionierermatrizen nur einmal zu Beginn aufgestellt werden müssen und dann zur Lösung aller 26 Probleme eingesetzt werden können, so dass der setup-Aufwand prozentual gesehen – zumindest bei den von uns gewählten Parameterkonfigurationen – kaum ins Gewicht fällt.

Bei *Euclid* gestaltet sich die Aufgabe der Suche nach der optimalen Parametereinstellung relativ einfach, da nur ein einziger Parameter – das *level* – zu untersuchen ist. In Tabelle 9, die den singulären Fall darstellt, wird deutlich, dass bei *level* 1 die wenigsten elementaren Rechenoperationen zur Lösungsberechnung anfallen, dicht gefolgt von *level* 2 (die Zahlen wurden mithilfe der Formeln des "operation count", Abschnitt 4, ermittelt). Höhere *levels* haben dagegen keine Chance

[12] Computertomographie
[13] Magnetic Resonance Imaging

level	0	1	2	3	4	5
$nz(F)$	482184	884096	1538350	2823602	4580018	6783930
#it Paar 1	118	78	60	46	37	30
#ops total ($\cdot 10^8$)	2.96	2.60	2.80	3.37	4.10	4.67
#it Paar 2	119	79	60	46	37	30
#ops total ($\cdot 10^8$)	2.99	2.63	2.80	3.37	4.10	4.67

Tabelle 9: Aufwandsabschätzung beim singulären Kopfproblem für Euclid, $\epsilon = 10^{-8}$

level	0	1	2	3
filter = 0%				
nz(G)	276507	880284	2126596	-
#it Paar 1	186	138	105	-
#ops total ($\cdot 10^8$)	4.79	6.92	10.56	-
#it Paar 2	187	138	100	-
#ops total ($\cdot 10^8$)	4.82	6.92	10.06	-
filter = 90%				
nz(G)	116914	129658	295499	508200
#it Paar 1	306	288	171	139
#ops total ($\cdot 10^8$)	5.91	5.71	4.54	4.88
#it Paar 2	308	290	172	140
#ops total ($\cdot 10^8$)	5.95	5.75	4.56	4.92

Tabelle 10: Aufwandsabschätzung beim singulären Kopfproblem für ParaSails, *thresh* = 0, $\epsilon = 10^{-8}$

mehr, da die Vorkonditionierungsmatrix hier bereits zu dicht besetzt ist, und auch die sinkenden Iterationenanzahlen den dadurch entstehenden Mehraufwand nicht mehr ausgleichen können.

Für den regularisierten Fall ergibt sich genau das gleiche Bild, wobei die Iterationenzahlen (und damit auch die Gesamtoperationenzahlen) allgemein um ca. 30% höher liegen. Für *level* 1 ergibt sich beispielsweise:

	#it	$nz(F)$	#ops total
Paar 1	102	884077	$3.39 \cdot 10^8$
Paar 2	100	884077	$3.33 \cdot 10^8$

Schwieriger wird das Herausfinden der besten Parameterkonfiguration dagegen bei *ParaSails*, da hier drei voneinander unabhängige Parameter zur Verfügung stehen, und sich somit zahlreiche Kombinationsmöglichkeiten ergeben.

Betrachten wir zunächst einmal den Fall mit *thresh* = 0 und *filter* = 0% bei singulärer Koeffizientenmatrix (Tabelle 10). Es wird hier ersichtlich, dass mit höheren *levels* die "Dichte" der Vorkonditionierungsmatrix im Verhältnis zu den sinkenden Iterationenzahlen viel zu stark ansteigt, und deshalb die Gesamtoperationenanzahl wächst (vergleiche auch Ergebnisse der Aufwandsabschätzung beim Varga-Problem, Abschnitt 9.2). Als Kandidat für die optimale Parametereinstellung kommt daher nur der Fall mit *level* = 0 ($\rightarrow$ SPAI-1) in Frage, wobei selbst dort die Operationenanzahl noch immer fast doppelt so groß ist wie der optimale Wert bei *Euclid*.

Bei einer regularisierten Matrix ergibt sich auch hier ein analoges Ergebnis mit um etwa 30% höheren Iterationenanzahlen. Bei *level* 0 beispielsweise:

	#it	$nz(G)$	#ops total
Paar 1	243	276502	$6.26 \cdot 10^8$
Paar 2	246	276502	$6.33 \cdot 10^8$

Widmen wir uns jetzt einmal dem Fall, wenn der *filter*-Wert auf 90% gesetzt wird. Tabelle 10 zeigt die Ergebnisse (singulärer Fall). Man sieht, dass diesmal das Optimum an Operationen erst bei *level* 2 erreicht wird [14], da vorher die Vorkonditionierungsmatrix durch den großen *filter*-Wert noch zu wenige von Null verschiedene Einträge enthält und damit zu ungenau ist. Für *level*-Werte größer als 2 dagegen wird G – trotz der Ausdünnung durch den Filter – im Verhältnis zur Iterationenanzahl bereits wieder zu dicht-besetzt, und die Gesamtoperationenanzahl steigt. Das Optimum bei *level* 2 liegt sogar noch etwas unter der bisherigen Bestmarke vom Fall mit *filter*= 0%. Interessant ist, dass die beiden dazugehörigen nz-Werte dabei ungefähr gleich sind, so dass diese Zahlen anscheinend den Bereich darstellen, wo die Balance zwischen einer gewissen Genauigkeit der Vorkonditionierungsmatrix einerseits, und einem dennoch akzeptablen Gesamtaufwand andererseits erreicht wird.

Für den wiederum analogen regularisierten Fall gilt für *level* 2:

	#it	$nz(G)$	#ops total
Paar 1	224	295494	$5.94 \cdot 10^8$
Paar 2	216	295494	$5.73 \cdot 10^8$

Bis jetzt war der *thresh*-Wert immer unter der “kritischen Stelle”, die bei unseren Vorwärtsproblemen genau zwischen 0.16 und 0.17 liegt: Bei einem *thresh* von 0.16 beträgt die nz-Zahl von G 239444 (87% der ursprünglichen 276507 Einträge bei *level* 0 und *filter* 0%), bei 0.17 dagegen nur noch 117687 (43%). Die Auswirkung des *thresh*-Wertes auf die “Dichte” von G (bei *level* 0 und *filter* 0%) wird auch noch einmal in der folgenden Tabelle deutlich:

thresh	$nz(G)$	*thresh*	$nz(G)$
0.40	70840	0.16	239444
0.25	76218	0.13	257023
0.20	91364	0.08	270654
0.17	117687	0.05	276507

Wählt man nun für *thresh* Werte größer oder gleich 0.17, ist das resultierende Matrixmuster zu dünn-besetzt und damit zu ungenau, so dass selbst höhere *levels* dies nicht mehr kompensieren können, und somit die daraus entstehenden Iterationenzahlen im Verhältnis zur Matrixdichte zu hoch sind, um an unsere optimalen Gesamtoperationenanzahlen heranzukommen. Beispielsweise ergibt sich für *thresh* = 0.2, *level* = 0 und *filter* = 0%:

	#it	$nz(G)$	#ops total
Paar 1	327	91364	$5.98 \cdot 10^8$
Paar 2	328	91364	$6.00 \cdot 10^8$

oder für *thresh* = 0.2, *level* = 3 und *filter* = 0% (bei beiden Paaren):

#it	$nz(G)$	#ops total
313	235437	$7.54 \cdot 10^8$

[14] Der nz-Wert von G *vor* der Filterung ist hier übrigens 2126552 (nz-ratio = 7.69); der zum *filter*-Wert von 90% gehörende Absolutwert beträgt 0.126, so dass also alle kleineren Werte in der Matrix entfernt werden.

Auch bei weiteren diversen Versuchen mit verschiedenen Kombinationen aus *thresh*, *level* und *filter*, die als eventuell erfolgsversprechend gesehen werden konnten, ergaben sich keine Gesamtoperationenzahlen, die unser bisheriges Optimum (bei *thresh* = 0, *level* = 2, *filter* = 90%) übertreffen konnten, wobei sich für den singulären und den regulären Fall genau die gleichen Verhältnisse herauskristallisiert haben. Deshalb kann man davon ausgehen, dass es höchstwahrscheinlich wirklich keine (zumindest deutlich) bessere Parameterkonfiguration gibt, wobei bei der Fülle von Kombinationsmöglichkeiten von verschiedenen Parameterwerten bei *ParaSails* dafür natürlich keinerlei Garantie übernommen werden kann.

Zusammenfassend lässt sich sagen, dass bei den bisherigen Parameteruntersuchungen einige "Faustregeln" deutlich wurden, die man beim Umgang mit *ParaSails* möglicherweise berücksichtigen sollte:

Zunächst darf das "pattern" von $\tilde{A}$ (also vor einer eventuellen Potenzierung, siehe auch Abschnitt 3.1) nicht zu dünn-besetzt sein, d. h. der *thresh*-Wert sollte so gewählt werden, dass ungefähr so viele Nichtnull-Einträge wie in der Koeffizientenmatrix A existieren, da eine gewisse Genauigkeit erforderlich ist, die bei zu dünn-besetzten "Anfangsmatrizen" auch durch größere *level*-Werte oft nicht mehr ausreichend zu erhöhen ist.

Hohe *levels* bringen im allgemeinen nichts, da die nz-Zahl von G dadurch überproportional stark ansteigt, so dass auch eventuelle hohe *filter*-Werte meist nur noch wenig daran ändern können, und sich außerdem die setup-Kosten von G deutlich vergrößern.

Erfolgreiche Vorkonditionierungsmatrizen müssen in etwa so dicht besetzt sein wie die Matrix A, weshalb man also entweder gleich direkt deren Muster verwendet (d. h. das SPAI-1-Verfahren mit *thresh* = 0, *level* = 0, *filter* = 0%), oder *thresh* so bestimmt, dass in $\tilde{A}$ nur *etwas* weniger oder genausoviele nz-Einträge vorhanden sind wie in A, dann niedrige *levels* (z. B. 1 oder 2) wählt (dadurch hält sich auch der setup-Aufwand in Grenzen) und abschließend mit dem *filter*-Wert in etwa so viele Einträge eliminiert, bis die Gesamtzahl an Nichtnullen wieder ungefähr der von A entspricht (also beispielsweise wie bei unserem "besten Kandidaten" mit *thresh* = 0, *level* = 2 und *filter* = 90%). Der Vorteil der letzteren Vorgehensweise besteht darin, dass sich durch das – aufgrund der zwar niedrigen, aber dennoch von Null verschiedenen *level*-Werte – dichter gewordene Besetztheitsmuster der Matrix $\tilde{A}^m$ eine größere Flexibilität für die Lösung der "least-squares"-Probleme (siehe Abschnitt 3.1) ergibt, und dadurch die Genauigkeit der approximativen Inversen erhöht wird. Aus dieser relativ genauen Matrix kann man dann mithilfe des Filters getrost wieder zahlreiche kleine Werte entfernen und damit viel zukünftigen Rechenaufwand einsparen, ohne jedoch viel von der Genauigkeit zu verlieren.

8.3 Testergebnisse

Jetzt, da wir die jeweils besten Parametereinstellungen unserer beiden Vorkonditionierer bestimmt haben, können wir die Lösung der 26 Vorwärtsprobleme angehen. Die Berechnungen sollen dabei auf drei verschiedenen Rechnerarchitekturen durchgeführt werden (siehe auch [16]):

- PIV: Pentium 4 Prozessor, Taktfrequenz 1500 MHz, Hauptspeicher 1024 MB; Betriebssystem Linux; Compiler/Linker gcc/g++ (2.95.2), Tuning Option -O3
- A21264: Alpha 21264 Prozessor, Taktfrequenz 500 MHz, Hauptspeicher 640 MB; Betriebssystem Compaq Tru64 4.0F; Compiler/Linker cc (DEC C V5.9-005), Tuning Optionen -fast -tune host
- Athlon: AMD Athlon Prozessor, Taktfrequenz 700 MHz, Hauptspeicher 768 MB; Betriebssystem Linux; Compiler/Linker gcc/g++ (2.95.2), Tuning Option -O3

Uns interessieren nun vor allem die Laufzeiten, die sich durch die Verwendung von *Euclid* und *ParaSails* (mit ihren jeweils optimalen Parameterwerten) insgesamt bei der Lösung aller 26 Probleme einschließlich der setup-Kosten der Vorkonditionierer (die ja prozentual gesehen in unserem Fall nur eine geringe Rolle spielen) ergeben, da natürlich letztlich für den Anwender immer der Zeitaufwand eines Lösungsvefahrens das entscheidende Argument ist.

	Matrix nicht-regularisiert			Matrix regularisiert		
Rechner	PIV	Athlon	A21264	PIV	Athlon	A21264
Euclid *level* = 1						
∅ Gesamtzeit	65.56	287.07	136.76	81.21	364.66	177.10
Std.abw.	0.15	5.00	0.17	0.12	0.51	0.31
Euclid *level* = 2						
∅ Gesamtzeit	66.44	300.00	144.60	83.07	382.00	182.30
Std.abw.	0.11	6.41	0.21	0.13	1.41	0.20
ParaSails *thresh* = 0, *level* = 2, *filter* = 90%						
∅ Gesamtzeit	116.34	528.12	225.8	143.21	637.45	289.8
Std.abw.	0.31	9.57	0.40	0.63	17.30	0.51
ParaSails *thresh* = 0, *level* = 0, *filter* = 0%						
∅ Gesamtzeit	117.67	-	-	145.87	-	-
Std.abw.	0.24	-	-	0.63	-	-

Tabelle 11: user-times (Gesamtlösezeit) beim Kopfproblem

Tabelle 11 zeigt die Ergebnisse. Die Zeitangaben entsprechen der user-time in Sekunden. In jedem Fall wurden jeweils zehn Programmdurchläufe gemessen und dann der Durchschnitt ermittelt. Dazu ist auch immer die resultierende Standardabweichung angegeben, wobei diese – wie leicht auffällt – durchwegs sehr kleine Werte aufweist (meist unter 1% des Durchschnittswertes), die praktisch zu vernachlässigen sind.

Die durchschnittlichen Zeitangaben der Laufzeit spiegeln genau das Ergebnis unserer vorherigen Aufwandsabschätzung wider: *Euclid* ist klar das schnellste Verfahren, wobei zwischen dem Optimum von *level* 1 und dem Resultat von *level* 2 nur ein geringer Unterschied besteht, wohingegen *ParaSails* in dessen bester Parameterkonfiguration fast doppelt so lange braucht.

Bei den Rechnern erweist sich der Pentium als mit Abstand am leistungsfähigsten, während der AMD Athlon weit abgeschlagen landet. Die Verhältnisse zwischen *Euclid* und *ParaSails*, und auch zwischen den verschiedenen Parametereinstellungen, sind jedoch auf allen Architekturen exakt die gleichen, weshalb es auch genügt, (interessehalber) die zweitbeste *ParaSails*-Konfiguration nur auf dem Pentium zu betrachten. Für *Euclid* ergibt sich darauf übrigens für *level* 0 eine Laufzeit von 81 s und für *level* 3 von 78 s – beide Zeiten sind also weit vom Optimum entfernt.

Auch der Mehraufwand für den regularisierten Fall ist bei den Laufzeiten wieder deutlich zu sehen: die Zeiten liegen überall um 20-30% höher als bei der singulären Koeffizientenmatrix, bei ansonsten jedoch genau gleichen Verhältnissen.

Abbildungen 9 und 10 zeigen noch einmal anschaulich die Daten von *Euclid* und *ParaSails* bei jeweils optimaler Parameterkonfiguration.

Betrachtet man die Laufzeiten des nicht-vorkonditionierten CG-Verfahrens auf dem Pentium von durchschnittlich 150.0 s (singulär) bzw. 183.9 s (regulär), so wird der Nutzen der Vorkonditionierung deutlich: *Euclid* spart ca. 55% an Zeit ein, *ParaSails* immerhin noch ca. 22%.

Was neben den Laufzeiten natürlich auch noch interessant ist, sind die von den einzelnen Verfahren zur Lösung benötigten Iterationenzahlen, die in Tabelle 12 dargestellt sind. Die Zahlen ergeben sich als Durchschnitt aus den 26 Werten für jedes einzelne Vorwärtsproblem. Die Standardabweichungen sind dabei relativ gering ($< 2\%$), d. h. die einzelnen Probleme unterscheiden sich hinsichtlich der jeweils resultierenden Iterationenanzahl nur wenig (was auch nicht verwundert, da ja schließlich die Koeffizientenmatrix bei allen 26 Vorwärtsproblemen die gleiche ist, und

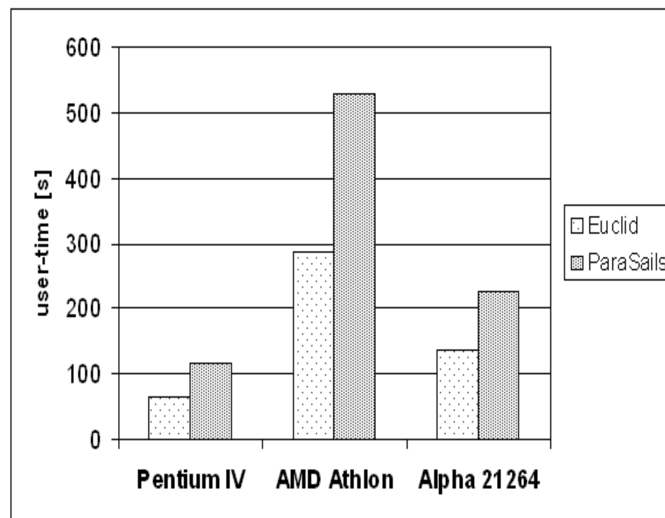


Abbildung 9: Gesamtlösezeiten für alle 26 Vorwärtsprobleme im *singulären* Fall mit *Euclid* und *ParaSails* bei optimaler Parametereinstellung

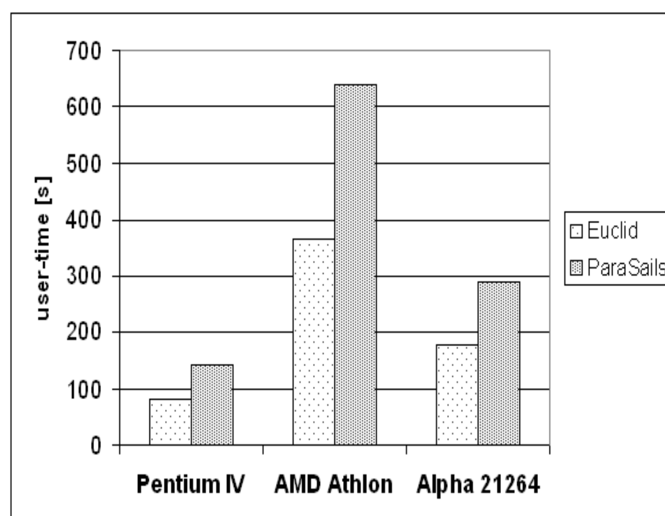


Abbildung 10: Gesamtlösezeiten für alle 26 Vorwärtsprobleme im *regularisierten* Fall mit *Euclid* und *ParaSails* bei optimaler Parametereinstellung

	CG	Euclid l=1	Euclid l=2	ParaSails th=0, l=2, f=90%	ParaSails th=0, l=0, f=0%
Matrix nicht-regularisiert					
∅ #it	401.5	78.7	59.8	171.2	187.5
Std.abw.	9.5	1.7	1.0	2.8	2.9
Matrix regularisiert					
∅ #it	527.7	102.4	77.6	223.0	245.3
Std.abw.	6.7	1.4	1.0	1.9	3.1

Tabelle 12: Durchschnittliche Iterationenzahlen bei der Lösung der 26 Vorwärtsprobleme

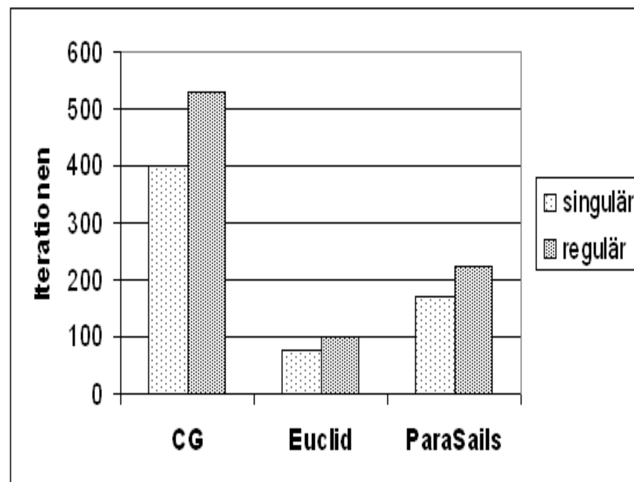


Abbildung 11: Durchschnittliche Iterationenzahlen bei der Lösung der Vorwärtsprobleme bei singulärer und regularisierter Koeffizientenmatrix

sich die rechten Seiten der LGS nur durch die unterschiedlichen Positionen der "1" und "-1" unterscheiden). Erwartungsgemäß weist *Euclid* wieder klar niedrigere Zahlen auf als *ParaSails*, beide liegen jedoch deutlich unter dem "reinen" CG-Verfahren (siehe auch Abbildung 11).

Abschließend noch eine Bemerkung zur Korrektheit der Lösung bei der Verwendung von *Euclid* im singulären Fall. Wie wir bereits gesehen haben, ist der Erstellungsvorgang der *Euclid*-Matrizen L und U nicht trivial, so dass gerade bei einer singulären Koeffizientenmatrix der Erfolg des Verfahrens nicht unbedingt garantiert ist. Deshalb liegt es nahe, die Korrektheit unserer mithilfe von *Euclid* erhaltenen Lösungen der Vorwärtsprobleme einmal zu prüfen.

Dazu muss zuerst die additive Konstante bestimmt werden, durch die sich die singuläre von der regulären Lösung unterscheidet. Dies erreicht man, indem man einfach ein ausgewähltes Voxel hernimmt und dessen regulären Lösungswert vom singulären abzieht (oder umgekehrt). Ist die Konstante bestimmt, wird nun der singuläre Lösungsvektor in gewisser Weise normiert, indem man die Konstante von allen Werten des Vektors abzieht (oder addiert, beim umgekehrten Fall). Dieser normierte Vektor der singulären Lösung sollte sich nun bei Korrektheit vom Lösungsvektor des regulären Falles praktisch nicht mehr unterscheiden.

In unserem Fall ergeben sich für die 2-Norm des Differenzvektors beider Lösungsvektoren bei allen 26 Vorwärtsproblemen Werte zwischen lediglich 10^{-15} und 10^{-16}, sowohl bei *level* 1, als auch bei *level* 2, so dass wir von einer korrekten Lösung der Vorwärtsprobleme auch im singulären Fall ausgehen können.

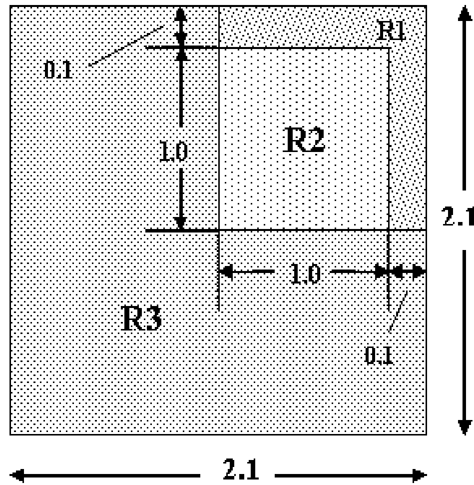


Abbildung 12: Gebiet mit den einzelnen Regionen beim Varga-Problem

9 Das Varga-Problem

9.1 Problembeschreibung

Zum Abschluss wollen wir noch ein weiteres Problem untersuchen, für dessen Lösung ein vorkonditioniertes CG-Verfahren geeignet ist (und somit dabei auch *Euclid* und *ParaSails* zum Einsatz kommen können), das sich jedoch insofern von den vorangegangenen Problemstellungen unterscheidet, als hier anstelle einer Poisson-Gleichung eine Helmholtz-Gleichung vorliegt. Es handelt sich um das von R. Varga in Appendix B von [18] beschriebene Problem aus der Reaktorphysik, das auch bei Meijerink und Van der Vorst in [13] und [14] als Beispiel dient.

Betrachtet werden soll die numerische Lösung der folgenden zweidimensionalen elliptischen partiellen Differenzialgleichung:

$$-(D(x,y))u_x)_x \quad - \quad (D(x,y)u_y)_y \quad + \quad \sigma(x,y)u(x,y) \quad = \quad 0 \; , \qquad (x,y) \in R,$$

wobei R das Quadrat $0 < x, y < 2.1$ ist (siehe Abbildung 12), mit den Randbedingungen

$$\frac{\partial u(x,y)}{\partial n} = 0 \; , \qquad (x,y) \in \partial R$$

Dieses Quadrat ist in drei verschiedene Regionen eingeteilt, in denen die Funktionen D und σ jeweils andere Werte annehmen. Dadurch werden verschiedene in der Realität vorkommende Materialien simuliert (siehe Abbildung 12). Die Funktionen D und σ sind folgendermaßen definiert [18]:

Region	$D(x,y)$	$\sigma(x,y)$
1	1.0	0.02
2	2.0	0.03
3	3.0	0.05

Um das Problem mithilfe eines iterativen Lösungsverfahrens für LGS zu lösen, muss es natürlich zunächst wieder diskretisiert werden, um eine “Fünf-Punkt-Stencil”-Approximation mit springenden Koeffizienten zu erhalten. Dazu wird die gesamte Fläche in Zellen eingeteilt (jede Zelle ist dabei Teil einer der drei Regionen), an deren Eckpunkten die zu ermittelnden Unbekannten sitzen

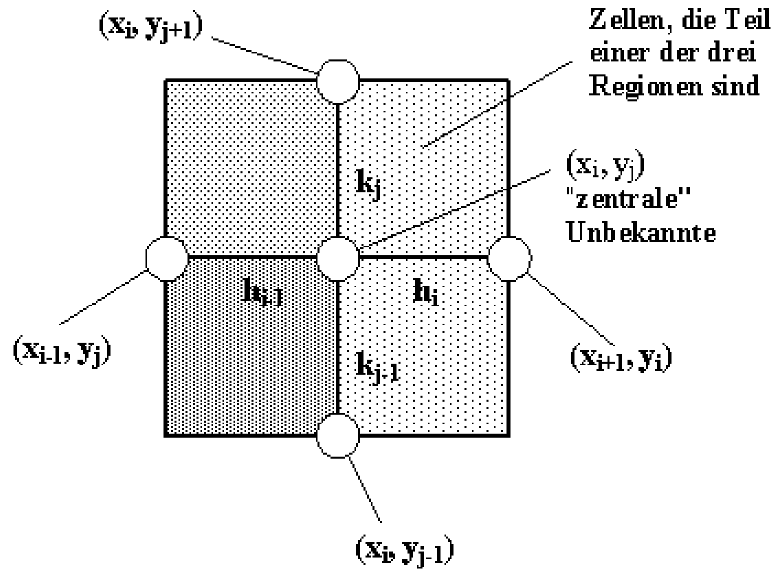


Abbildung 13: Stencil mit den vier Nachbarn und den Regionen beim Varga-Problem

(siehe Abbildung 13). Anders als bei den vorangegangenen Problemen mit springenden Koeffizienten kann man hier also den Unbekannten-Gitterpunkt nicht eindeutig einer bestimmten Region zuordnen. Eine Unbekannte (i, j) im Inneren des Gebietes R hat immer vier Nachbarn, deren Entfernungen von (i, j) mit h_{i-1}, h_i, k_{j-1} und k_j spezifiziert sind (siehe Abbildung 13).

Der Diskretisierung liegt nun die folgende Approximation zugrunde, was im Prinzip einer "Box Integration" entspricht:

$$\begin{aligned}
-\int\limits_{c_{ij}} (Du_x dy - Du_y dx) \quad \approx \quad & \left(\frac{k_{j-1}}{2} \cdot D_{i+\frac{1}{2},j-\frac{1}{2}} \; + \; \frac{k_j}{2} \cdot D_{i+\frac{1}{2},j+\frac{1}{2}} \right) \cdot \frac{u_{ij} - u_{i+1,j}}{h_i} \\
+ \; & \left(\frac{k_{j-1}}{2} \cdot D_{i-\frac{1}{2},j-\frac{1}{2}} \; + \; \frac{k_j}{2} \cdot D_{i-\frac{1}{2},j+\frac{1}{2}} \right) \cdot \frac{u_{ij} - u_{i-1,j}}{h_{i-1}} \\
+ \; & \left(\frac{h_{i-1}}{2} \cdot D_{i-\frac{1}{2},j+\frac{1}{2}} \; + \; \frac{h_i}{2} \cdot D_{i+\frac{1}{2},j+\frac{1}{2}} \right) \cdot \frac{u_{ij} - u_{i,j+1}}{k_j} \\
+ \; & \left(\frac{h_{i-1}}{2} \cdot D_{i-\frac{1}{2},j-\frac{1}{2}} \; + \; \frac{h_i}{2} \cdot D_{i+\frac{1}{2},j-\frac{1}{2}} \right) \cdot \frac{u_{ij} - u_{i,j-1}}{k_{j-1}} \quad ,
\end{aligned}$$

wobei c_{ij} die Grenzlinie der "Integrations-Box" um den Punkt (i, j) ist, und $D_{i+\frac{1}{2},j+\frac{1}{2}} \equiv D(x_i + \frac{1}{2}h_i,\; y_j + \frac{1}{2}k_j)$, usw.

Aus dieser Approximierung lassen sich nun die Koeffizienten der vier Nachbarelemente der Unbekannten (i, j) leicht bestimmen:

$$R \;=\; -\left(\frac{k_{j-1}}{2h_i} \cdot D_{i+\frac{1}{2},j-\frac{1}{2}} \; + \; \frac{k_j}{2h_i} \cdot D_{i+\frac{1}{2},j+\frac{1}{2}} \right) \qquad \text{Koeffizient des rechten Nachbarn}$$

$$L \;=\; -\left(\frac{k_{j-1}}{2h_{i-1}} \cdot D_{i-\frac{1}{2},j-\frac{1}{2}} \; + \; \frac{k_j}{2h_{i-1}} \cdot D_{i-\frac{1}{2},j+\frac{1}{2}} \right) \qquad \text{Koeffizient des linken Nachbarn}$$

$$T \;=\; -\left(\frac{h_{i-1}}{2k_j} \cdot D_{i-\frac{1}{2},j+\frac{1}{2}} \; + \; \frac{h_i}{2k_j} \cdot D_{i+\frac{1}{2},j+\frac{1}{2}} \right) \qquad \text{Koeffizient des oberen Nachbarn}$$

$$B = -\left(\frac{h_{i-1}}{2k_{j-1}} \cdot D_{i-\frac{1}{2},j-\frac{1}{2}} + \frac{h_i}{2k_{j-1}} \cdot D_{i+\frac{1}{2},j-\frac{1}{2}}\right) \quad \text{Koeffizient des unteren Nachbarn}$$

Um den Koeffizienten der "zentralen" Unbekannten zu bestimmen, muss nun noch die Funktion σ mit einbezogen werden. Hierbei sind die σ-Werte aus den vier Nachbarzellen der Unbekannten entscheidend. Da σ in jeder Zelle einen konstanten Wert besitzt, lässt sich der exakte Wert des Integrals angeben. Es ergibt sich die folgende Formel:

$$\begin{aligned} C &= -(R+L+T+B) \\ &+ \sigma_{i-\frac{1}{2},j-\frac{1}{2}} \cdot \frac{k_{j-1}h_{i-1}}{4} + \sigma_{i+\frac{1}{2},j-\frac{1}{2}} \cdot \frac{k_{j-1}h_i}{4} + \sigma_{i-\frac{1}{2},j+\frac{1}{2}} \cdot \frac{k_j h_{i-1}}{4} + \sigma_{i+\frac{1}{2},j+\frac{1}{2}} \cdot \frac{k_j h_i}{4} \end{aligned}$$

In unserem Fall haben wir das gesamte Gebiet (also das 2.1 x 2.1 - Quadrat) in lauter *gleichgroße* Zellen mit der Maschenweite 0.05 eingeteilt (wie in [13] und [14]), so dass sich die obigen Formeln vereinfachen und die einzelnen Koeffizientenwerte dadurch auf einfachen Durchschnittsberechnungen basieren:

$$\begin{aligned} R &= -\frac{1}{2} \cdot (D_{i+\frac{1}{2},j-\frac{1}{2}} + D_{i+\frac{1}{2},j+\frac{1}{2}})\,, \quad L = -\frac{1}{2} \cdot (D_{i-\frac{1}{2},j-\frac{1}{2}} + D_{i-\frac{1}{2},j+\frac{1}{2}})\,, \\ T &= -\frac{1}{2} \cdot (D_{i-\frac{1}{2},j+\frac{1}{2}} + D_{i+\frac{1}{2},j+\frac{1}{2}})\,, \quad B = -\frac{1}{2} \cdot (D_{i-\frac{1}{2},j-\frac{1}{2}} + D_{i+\frac{1}{2},j-\frac{1}{2}})\,, \\ C &= -(R+L+T+B) + \frac{0.05^2}{4} \cdot (\sigma_{i-\frac{1}{2},j-\frac{1}{2}} + \sigma_{i+\frac{1}{2},j-\frac{1}{2}} + \sigma_{i-\frac{1}{2},j+\frac{1}{2}} + \sigma_{i+\frac{1}{2},j+\frac{1}{2}}) \end{aligned}$$

Jetzt können wir die Koeffizientenmatrix A aufstellen und dann das LGS $Ax = 0$ lösen.

Aus der gewählten Maschenweite von 0.05 ergeben sich für jede Dimension 42 Zellen, so dass insgesamt $43^2 = 1849$ Unbekannte existieren, d. h. A ist eine 1849 x 1849 - Matrix. Sie besitzt 9073 Nichtnull-Einträge, ist also dünn-besetzt ("sparsity ratio": 0.27%).

Natürlich sind nun wieder die besonderen Eigenschaften dieser Matrix von Interesse: A ist eine reelle Matrix und hat – da D und σ positiv sind – positive Diagonaleinträge und nicht-positive Außer-Diagonaleinträge. Da $\sigma > 0$, ist sie streng diagonal-dominant und dadurch regulär. Aufgrund der auch hier geltenden Tatsache, dass der Koeffizient, der eine Unbekannte k mit einer Unbekannten l verbindet, identisch ist mit dem Koeffizienten, der l mit k verbindet, ist A zudem symmetrisch. Also ist A auch positiv-definit. Damit hat die Matrix alle Eigenschaften, die für die Lösung mithilfe des (vorkonditionierten) CG-Verfahrens nötig sind.

Natürlich ist in unserem speziellen Fall des Problems die Lösung bereits bekannt, da bei uns die rechte Seite der Gleichung gleich Null ist,[15] und deshalb auch die Lösung gleich Null sein muss (d. h. der Lösungsvektor ist der Nullvektor). Dennoch wollen wir das PCG-Verfahren mit *Euclid* und *ParaSails* zur Lösungsbestimmung einsetzen, um vor allem Aufschlüsse über die Konvergenz zu erhalten.

9.2 Testergebnisse

Zur Untersuchung der Konvergenz bei der Lösung des Varga-Problems werden bei uns in Analogie zu [13] und [14] vier verschiedene Toleranzwerte ϵ für das relative Residuum in der 2-Norm als Abbruchkriterium betrachtet: 10^{-3}, 10^{-6}, 10^{-9} und 10^{-12}. Tabelle 13 zeigt die Ergebnisse für das nicht-vorkonditionierte CG-Verfahren und für *Euclid* und *ParaSails* (*thresh* $= 0.1$, d. h. kleiner als der "kritische Bereich") bei zwei unterschiedlichen Startvektoren. Die Lösezeiten sind hier diesmal nicht berücksichtigt, da bei einer Problemgröße von nur 1849 Unbekannten die Zeiten keine Rolle spielen.

Erwartungsgemäß lässt sich erkennen, dass mit kleiner werdenden ϵ-Werten die Iterationenanzahl für alle Verfahren relativ gleichmäßig (nahezu linear) ansteigt, wobei *Euclid* insgesamt wieder bei weitem die geringsten Zahlen aufweist, *ParaSails* in etwa in der Mitte liegt, und das "reine" CG-Verfahren am schlechtesten konvergiert (siehe auch Abbildung 14). Höhere *levels* – sowohl

[15]Normalerweise ergibt sich die rechte Seite des LGS beim Varga-Problem aus einer von Null verschiedenen Funktion $S(x, y)$

	Startvektor $\vec{10^4}$				Startvektor zufällig			
Toleranz	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-3}	10^{-6}	10^{-9}	10^{-12}
CG								
	197	229	259	398	385	416	580	622
Euclid								
level 1	39	49	55	77	64	70	99	107
level 9	11	13	16	22	18	21	29	31
ParaSails, *filter* = 0%								
level 0	88	103	116	128	175	189	264	282
level 5	34	43	49	66	55	60	84	91
ParaSails, *filter* = 90%								
level 0	161	187	208	302	315	343	461	504
level 5	88	105	122	164	138	149	207	226

Tabelle 13: Konvergenz beim Varga-Problem ($thresh = 0.1$ bei ParaSails)

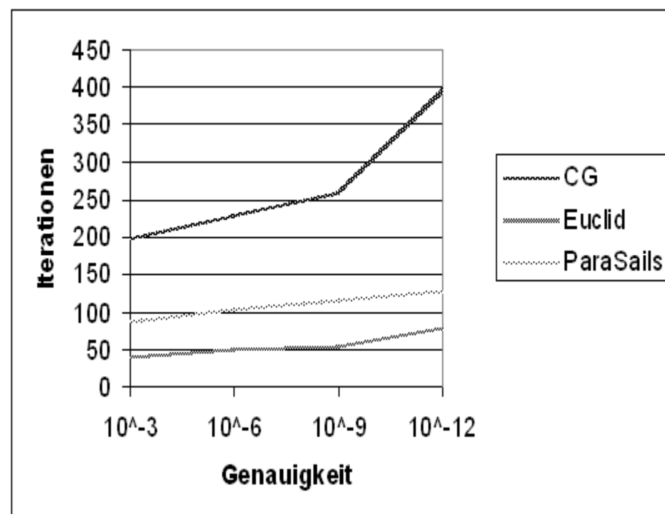


Abbildung 14: Iterationen in Abhängigkeit von der Toleranzgrenze ϵ beim Varga-Problem, Startvektor $\vec{10^4}$ (Euclid: $level = 1$, ParaSails: $thresh < 0.2$, $level = 0$, $filter = 0\%$)

			Toleranz 10^{-3}		Toleranz 10^{-6}	
level	$nz(F)$	#ops/it	#it	#ops total	#it	#ops total
0	9073	51084	55	2844063	75	3865743
1	12601	58140	37	2192679	46	2715939
3	22849	78636	20	1634715	28	2263803
5	35953	104844	14	1556019	19	2080239

Tabelle 14: Aufwandsabschätzung für Euclid beim Varga-Problem, Startvektor zufällig zwischen 0 und 1

bei *ParaSails*, als auch bei *Euclid* – verringern die benötigten Iterationen. Auch die Rolle des Startvektors wird beim Varga-Problem deutlich: Ein einheitlicher Startvektor mit lauter gleichen Einträgen (bei uns 10^4, wie bei Varga in [18]) ermöglicht bei einer zu erreichenden Lösung von $\vec{0}$ eine schnellere Konvergenz als ein mit zufällig ausgesuchten Werten besetzter.

Interessant ist die Tatsache, dass selbst bei einer sehr niedrig gewählten Toleranzgrenze ϵ, z. B. 10^{-100}, sich die Werte des Lösungsvektors, der ja eigentlich exakt $\vec{0}$ ergeben müsste, bei etwa 10^{-12} bis 10^{-11} einpendeln, wobei zwischen dem vorkonditionierten und dem nicht-vorkonditionierten Fall kein nennenswerter Unterschied zu erkennen ist. Vereinfacht man das Gleichungssystem, indem man $D(x, y)$ konstant auf 1 setzt, und somit keine springenden Koeffizienten mehr vorhanden sind, verbessern sich die Lösungswerte zwar auf ca. 10^{-14} bis 10^{-13}, aber auch hier wird die eigentliche Lösung nicht genau erreicht. Allerdings verringern sich die benötigten Iterationen (bei $\epsilon = 10^{-100}$ von z. B. 2400 auf 2000 beim CG-Verfahren, und von 480 auf 400 bei *Euclid*).

Setzt man zusätzlich auch noch die σ-Werte auf 0, so dass eine singuläre Koeffizientenmatrix entsteht, ergeben sich – sowohl beim “reinen” CG-Verfahren als auch beim vorkonditionierten – als Lösung “Ebenen”, die parallel zur xy-Ebene (“Nullebene”) liegen (z. B. eine Ebene im Abstand von 18.58 bei CG und 2307.43 bei *Euclid*). Hier unterscheiden sich die Lösungen also von der tatsächlichen durch eine additive Konstante.

Mithilfe von *MatLab* ergab sich für den größten Eigenwert λ_{max} der Koeffizientenmatrix des Varga-Problems ein Wert von 23.9333, während $\lambda_{min} = 1.0554 \cdot 10^{-5}$, so dass die Konditionszahl κ $2.27 \cdot 10^6$ beträgt, was eigentlich – hinsichtlich einer zu negativen Auswirkung auf die Konvergenz des CG-Verfahrens – nicht als übermäßig hoch erscheint. Dennoch müssen wir uns wohl damit abfinden, dass die wirklich exakte Lösung von $\vec{0}$ beim Varga-Problem von den von uns verwendeten Verfahren nicht erreicht wird.

Damit die Ergebnisse des “operation count” (siehe Abschnitt 4) für CG, *Euclid* und *ParaSails* auch hier zum Einsatz kommen, haben wir im nächsten “Experiment” eine Aufwandsabschätzung für die verschiedenen Verfahren bezüglich der Lösung des Varga-Problems vorgenommen [16], wobei einige ganz interessante Erkenntnisse gewonnen werden konnten (vergleiche auch Aufwandsabschätzung beim Kopfproblem, Abschnitt 8.2).

Der Startvektor ist diesmal mit zufälligen Werten zwischen 0 und 1 besetzt (wie bei Meijerink und Van der Vorst in [14]), und es werden die ϵ-Werte 10^{-3} und 10^{-6} betrachtet. Die Ergebnisse sind in den Tabellen 14 und 15 dargestellt. Die Tabellen enthalten u. a. die Daten, die für die Formeln des “operation count” (siehe Abschnitt 4) wichtig sind, also die Anzahl der benötigten Iterationen und die Anzahl der Nichtnull-Einträge der Fill-Matrix F, $nz(F)$, bei *Euclid* (siehe Abschnitt 3.2) bzw. der unteren Dreiecksmatrix G, $nz(G)$, bei *ParaSails* (siehe Abschnitt 3.1). Daraus ergeben sich die Beträge der pro Iteration erforderlichen elementaren Rechenoperationen (Additionen und Multiplikationen) “ops/it”, aus denen schließlich durch eine Multiplikation mit der Iterationenanzahl (zuzüglich der Initialisierungskosten) die insgesamt zur Lösung nötigen Operationen “ops total” ermittelt werden können.

[16] Die setup-Kosten für die Vorkonditionierungsmatrizen bleiben hier natürlich unberücksichtigt.

			Toleranz 10^{-3}		Toleranz 10^{-6}	
level	$nz(G)$	#ops/it	#it	#ops total	#it	#ops total
filter = 0%						
0	5461	52933	99	5276659	130	6917582
3	36279	176205	44	7912584	57	10203249
5	71821	318373	33	10808041	41	13355025
filter = 90%						
0	2109	39525	177	7018809	234	9271734
3	5777	54197	97	5294665	124	6757984
5	8509	65125	80	5258484	102	6691234

Tabelle 15: Aufwandsabschätzung für ParaSails (*thresh* = 0.1) beim Varga-Problem, Startvektor zufällig zwischen 0 und 1

Betrachten wir Tabelle 14, die die Ergebnisse von *Euclid* darstellt. Neben den aufgeführten Operationenzahlen sind hier zunächst auch die Iterationenzahlen von Interesse, da sich diese mit den von Meijerink und Van der Vorst ermittelten Zahlen aus Table II in [14] vergleichen lassen. Auch dort wird das Varga-Problem mithilfe eines vorkonditionierten CG-Verfahrens – bei einem Startvektor mit zufällig gewählten Einträgen zwischen 0 und 1 – gelöst, wobei die Vorkonditionierungsmatrix ebenfalls durch eine inkomplette LU-Zerlegung von A (wie bei *Euclid*) erstellt wird. So entspricht beispielsweise das in [14] erläuterte und verwendete ICCG(1,2)-Verfahren dem *Euclid*-Verfahren bei *level* 1, weshalb in diesem Fall auch die Iterationenzahlen praktisch identisch sind (in Table II in [14]: 38 für $\epsilon = 10^{-3}$ und 47 für $\epsilon = 10^{-6}$).

Nun aber zurück zur Aufwandsabschätzung. In Tabelle 14 wird deutlich, dass bei *Euclid* mit jedem höheren *level* die Gesamtoperationenanzahl sinkt und sich schließlich im Bereich von *level* 3 bis 5 bei einem gewissen Wert einpendelt. Die Zahl der Nichtnull-Einträge von F – und damit die pro Iteration benötigten Operationen – steigen mit wachsendem *level* relativ gleichmäßig und fast linear an.

Bei *ParaSails* hingegen (Tabelle 15) wird beim Fall mit *filter* = 0% klar, wie stark die "Dichte" von G mit höheren *levels* zunimmt, wodurch sich natürlich dann so große Zahlen an pro Iteration benötigten Operationen ergeben, dass die resultierenden Iterationenanzahlen zur Lösungsberechnung – auch wenn sie mit steigendem *level* sinken – diesen überproportionalen Mehraufwand nicht mehr kompensieren können, und deshalb die Gesamtoperationenanzahl mit jedem größeren *level* wächst.

Der "Explosion" der Besetztheits-Dichte von G kann man jedoch beispielsweise mit einem hohen (prozentualen) *filter*-Wert (wie bei uns mit 90%) entgegenwirken, was auch in Tabelle 15 ersichtlich ist. Man sieht, dass nun die nz-Anzahl von G – und damit die pro Iteration anfallenden Operationen – nur noch annähernd linear ansteigen, und somit durch die im Verhältnis stärker sinkenden Iterationenzahlen bei wachsendem level eine Verringerung der Gesamtoperationen erzielt werden kann, wobei sich die Zahl auch hier - wie bei *Euclid* – zwischen *level* 3 und 5 bei einem bestimmten Wert einpendelt. Interessant ist jedoch, dass die sich bei der Einstellung mit *level* = 0 und *filter* = 0% ergebende Gesamtoperationenanzahl nur hauchdünn verbessert werden kann, was wiederum unsere bereits erwähnte Vermutung bestätigt, dass bei *ParaSails* mit gleichzeitig niedrigen *level*- und *filter*-Werten (Prozentwert) zumeist die besten Ergebnisse erzielt werden können.

Vergleicht man beide Vorkonditionierer, so wird erneut die klare Überlegenheit von *Euclid* deutlich: 1556019 Operationen stehen in unserem Fall 5258484 Operationen bei *ParaSails* gegenüber (bei $\epsilon = 10^{-3}$). Verglichen mit dem nicht-vorkonditionierten CG-Verfahren ist jedoch auch *ParaSails* deutlich im Vorteil, wie man in der folgenden Tabelle erkennen kann, die die entsprechenden Daten

des CG-Verfahrens (bei einem – wie vorher – mit zufälligen Werten zwischen 0 und 1 besetzten Startvektor) enthält:

Toleranz	10^{-3}	10^{-6}
$nz(A)$	9073	9073
#ops/it	34787	34787
#it	239	302
#ops total	8332239	10523820

Zum Schluss noch ein Blick auf den vielleicht auch interessanten Fall, dass der *thresh*-Wert bei *ParaSails* größer ist als der "kritische Bereich", der auch beim Varga-Problem bei ≈ 0.25 liegt. Für *thresh* $= 0.4$ ergibt sich für das daraus resultierende SPAI-0-Verfahren (bei gleichem Startvektor wie zuvor):

Toleranz	10^{-3}	10^{-6}
$nz(G)$	1849	1849
#ops/it	38485	38485
#it	187	244
#ops total	7218539	9412184

Aufgrund des Diagonalmusters von G entspricht hier die nz-Zahl von G erwartungsgemäß genau der Anzahl der Unbekannten des Gleichungssystems.

Man sieht, dass diese "Konfiguration", bei der die *level*- und *filter*-Werte keine Rolle spielen, weit vom möglichen Optimum entfernt ist, da die Vorkonditionierungsmatrix zu ungenau ist.

Zusammenfassende Bemerkungen

Rückblickend auf unsere diversen Tests, Messungen und Untersuchungen hinsichtlich des Verhaltens und der Eigenschaften der *hypre*-Vorkonditionierer *Euclid* und *ParaSails* wurden doch einige Charakteristika und Auffälligkeiten deutlich.

Die wichtigste Erkenntnis ist wohl die Tatsache, dass *Euclid* in sämtlichen zwei- und dreidimensionalen Problemfällen die klar besseren Messergebnisse geliefert hat als *ParaSails* – sowohl was die Konvergenz angeht, als auch bezüglich des Zeitaufwandes bzw. der Anzahl der benötigten arithmetischen Operationen zur Lösungsberechnung des jeweiligen Problems. Außerdem zeigte *ParaSails* v. a. bei höheren *levels* unverhältnismäßig hohe setup-Kosten für das Aufstellen der Vorkonditionierungsmatrix.

Es ist jedoch zu vermuten, dass *ParaSails* bei einer Anwendung auf einem Parallelrechner – dafür wurde es ja auch ursprünglich entwickelt – deutlich besser abschneidet als im sequenziellen Fall. Wie in Abschnitt 3.1 gesehen, ergeben sich im Gegensatz zu *Euclid* aus der Definition des Verfahrens zahlreiche leicht zu parallelisierende Rechenschritte, so dass dadurch bei entsprechender Aufteilung auf parallele Prozessoren, sehr viel an Zeit eingespart werden kann.

Auch die "Bedienungsfreundlichkeit" spricht für *Euclid*, wo sich die Situation mit lediglich einem Parameter recht übersichtlich darstellt, wohingegen die drei praktisch voneinander unabhängigen Parameter bei *ParaSails* bzw. deren zahlreiche Kombinationsmöglichkeiten für teilweise unnötige Komplexität und Verwirrung sorgen, wenngleich der Benutzer so natürlich in der Lage ist, mehr Einfluss auf die genaue Gestalt der Vorkonditionierungsmatrix zu nehmen.

Allerdings ist das grundsätzliche Verfahren des setups bei *ParaSails* leichter verständlich und nachvollziehbar als bei *Euclid*, wo zudem die Gefahr besteht, dass eine Lösungsberechnung – gerade z. B. bei singulären Koeffizientenmatrizen – auch einmal nicht erfolgreich zu Ende gebracht wird, obwohl sich das Verfahren bei unseren Tests als sehr stabil erwiesen hat.

Was sicherlich darüber hinaus ersichtlich wurde, ist der allgemeine Nutzen – messbar in der Einsparung an Lösezeit bzw. an insgesamt benötigten elementaren Operationen – einer Vorkonditionierung beim CG-Verfahren, der umso größer ist, je mehr Unbekannte das zu lösende LGS

besitzt, da dann der “setup-overhead” immer weniger ins Gewicht fällt, und vor allem der Mehraufwand pro Iteration durch zumeist deutlich niedrigere Gesamtiterationenzahlen als beim nichtvorkonditionierten Fall mehr als kompensiert werden kann.

Wenn – wie beim 3D-Kopfproblem – dieselbe Vorkonditionierungsmatrix M nur einmal erstellt werden muss und anschließend für mehrere Lösungsvorgänge benützt werden kann, lässt sich der Anteil der setup-Kosten für M an den Gesamtkosten besonders gering halten.

Wie wir beim Lösen der Vorwärtsprobleme des Kopfproblems gesehen haben, können durch die Vorkonditionierung Zeiteinsparungen von mehr als 50% (mit *Euclid*) im Vergleich zum “reinen” CG-Verfahren erzielt werden.

Eines wurde allerdings auch klar: Der beste Vorkonditionierer ist immer nur so gut wie der Rechner, auf dem er angewandt wird, d. h. mindestens genauso wichtig wie ein effektiver und effizienter Vorkonditionierungsalgorithmus ist – v. a. bei sehr großen Problemen – die Wahl eines leistungsstarken und schnellen Computers zur Lösungsberechnung. Erst die Kombination aus beiden Faktoren ermöglicht wirklich optimale Ergebnisse.

Literatur

[1] R. Barret, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Rousine und H. Van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, S. 5-45, SIAM, Philadelphia, 1994

[2] O. Bröker und M. J. Grote, Sparse Approximate Inverse Smoothers For Geometric and Algebraic Multigrid, S. 1-6, Departement Informatik, ETH Zürich, 2000

[3] E. Chow, Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns, Intl. J. High Perf. Comput. Appl., 15, S. 56-74, 2001

[4] E. Chow, ParaSails (Parallel sparse approximate inverse (least-squares) preconditioner) User's Guide, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2001

[5] R. D. Falgout, J. E. Jones, und U. M. Yang, Conceptual Interfaces in hypre, Lawrence Livermore National Laboratory technical report UCRL-JC-148957, Juli 2002

[6] R. D. Falgout, und U. M. Yang, hypre: a Library of High Performance Preconditioners, in: Computational Science – ICCS 2002 Part III, P.M.A. Sloot, C.J.K. Tan. J.J. Dongarra, und A.G. Hoekstra, Hrsg., Lecture Notes in Computer Science, vol. 2331, S. 632-641, Springer Verlag, 2002. Auch erhältlich als Lawrence Livermore National Laboratory technical report UCRL-JC-146175

[7] HYPRE homepage: http://www.llnl.gov/CASC/hypre

[8] HYPRE (high performance preconditioners) Reference Manual, Version 1.6.0, Lawrence Livermore National Laboratory, 2001

[9] HYPRE User's Manual, Software Version 1.6.0, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2001

[10] D. Hysom und A. Pothen, A Scalable Parallel Algorithm For Incomplete Factor Preconditioning, SIAM J. Sci. Comput., Vol. 22, No. 6, S. 2194-2215

[11] D. Hysom und A. Pothen, Euclid User Manual (A Scalable ILU Preconditioning Library for the Parallel Solution of Sparse Linear Systems), Old Dominion University, Norfolk, VA, und Lawrence Livermore National Laboratory, 2001

[12] L. Y. Kolotilina und A. Y. Yeremin, Factorized Sparse Approximate Inverse Preconditionings I. Theory, SIAM J. Matrix Anal. Appl. Vol. 14, No. 1, S. 45-58, 1993

[13] J. A. Meijerink und H. A. Van der Vorst, An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix, Mathematics of Computation, Vol. 31, No. 137, S. 148-162, 1977

[14] J. A. Meijerink und H. A. Van der Vorst, Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations as They Occur in Practical Problems, Journal of Computational Physics 44, S. 134-155, 1981

[15] A. R. Mitchell und D. F. Griffiths, The Finite Difference Method in Partial Differential Equations, John Wiley & Sons, 3. Ed., 1987

[16] M. Mohr, Comparison of Solvers for a Bioelectric Field Problem, Lehrstuhlbericht 01-2, Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, 2001

[17] J. R. Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, Edition $1\frac{1}{4}$, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994

[18] R. Varga, Matrix Iterative Analysis, Series in Automatic Computation, Prentice Hall, 1962