Institut

für

Mechanik

UNIKASSEL
VERSITÄT

Vergleichende Studien bei der iterativen Gleichungslösung im Rahmen der nichtlinearen Methode der finiten Elemente

Rui Chen
Matr.-Nr.22550127

Institute of Mechanics
University of Kassel

Mitteilung des Instituts für Mechanik Nr.12/2006
Report of the Institute of Mechanics No.12/2006

Herausgeber/editor

Der Geschäftsführende Direktor
Institut für Mechanik
Universität Kassel

Organisation und Verwaltung/organization and administration

Priv.-Doz. Dr.-Ing. Stefan Hartmann
Institut für Mechanik
Universität Kassel
Mönchebergstr.7
34125 Kassel
Germany

UNIKASSEL
VERSITÄT

Diplomarbeit I

Vergleichende Studien bei der iterativen Gleichungslösung im Rahmen der nichtlinearen Methode der finiten Elemente

Rui Chen
Matr.-Nr. 22550127

Universität Kassel
Fachbereich Maschinenbau
Fachgebiet Kontinuumsmechanik

August 2006

1. Betreuer und Prüfer: PD Dr.-Ing. habil. Stefan Hartmann
2. Betreuer und Prüfer: Prof. Dr. rer. nat. Andreas Meister

Vorwort

Ein besonderes Dankeschön an Herrn PD Dr.-Ing. S. Hartmann, der mir dieses interessante Thema gegeben und eine sehr hilfreiche Betreuung angeboten hat.

Mein Dank gilt ebenfalls Herrn Prof. Dr. A. Meister für die Übernahme als Gutachter sowie seinen konstruktiven Vorschlägen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der Einleitung wird zunächst die Aufgabenstellung für die Diplomarbeit vorgestellt. Danach wird einen kurzen Überblick über die zu implementierenden modernen Verfahren gegeben. Abschließend wird die Gliederung dieser Arbeit kurz erläutert.

1.1 Aufgabenstellung

Die Anwendung von Finite-Elementen Methoden bei der Simulation von Bauteilen mit nichtlinearen Materialeigenschaften führt auf nichtlineare Gleichungssysteme. Zur Lösung solcher nichtlinaren Gleichungssysteme wird oft das Newton Verfahren eingesetzt, wodurch lineare Gleichungssysteme ergibt:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.1}$$

Hierzu gelten $\mathbf{A} \in \mathbb{R}^{n \times n}$ und $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. Die Matrix $\mathbf{A}$ repräsentiert die Struktursteifigkeit und stellt oft eine große und schwach-besetzte Matrix dar. Ein wesentlicher Punkt in einem Finite-Elemente Programm stellt somit die effiziente Lösung großer schwachbesetzter linearer Gleichungssysteme dar. Die üblichen direkten Lösungsverfahren, wie z.B. das Gauss-Eliminationsverfahren, bieten eine effiziente Vorgehensweise bis zu gewissen Größenordnung (ca. 30000 Unbekannte) der auftretenden Gleichungssysteme. Danach stellen die iterativen Gleichungslöser aus Gründen der Rechenzeit und Speicherplatzbelegung die Verfahren der Wahl dar (siehe [1], S. 168 ff).

Unter der Vielzahl von modernen iterativen Verfahren haben sich insbesondere ein vorkonditioniertes BiCGStab-Verfahren für unsymmetrische Systeme und ein vorkonditioniertes CG-Verfahren für symmetrische Systeme als effiziente Vorgehensweise herausgestellt. Dabei bedingt die Vorkonditionierung eine wesentliche Voraussetzung für die

schnelle Konvergenz. Zwei wesentliche Aufgaben stehen bei der Implementation an. Erstens, die Präkonditionierung und, zweitens, die Matrix-Vektor Multiplikation, die bei finiten Elementen aufgrund der Schwachbesetztheit der Matrix speziell an die Speichertechnik angepasst werden muss.

In dieser Diplomarbeit sollen zwei vorkonditionierten konjugierten Gradientenlöser (CG und BiCGStab) jeweils für symmetrische und unsymmetrische Matrizen in einen am *Institut für Mechanik* vorliegenden Finite-Elemente Code (TASA-FEM, Version 1.0, siehe [14]) implementiert werden. Hierzu sind für das CG-Verfahren ein symmetrisches Gauss-Seidel Verfahren und eine unvollständige Cholesky-Zerlegung als Vorkonditionierer heranzuziehen und für das BiCGStab-Verfahren eine unvollständige LU-Zerlegung einzusetzen. Die Verfahren sollen dann mit dem bereits implementierten UMFPACK-Löser bei unterschiedlichen Vernetzungsdichten sowie variierender Kompressibilität des zugrundeliegenden Materialmodells der finiten Hyperelastizität untersucht werden.

1.2 Stand der Technik

Die iterativen Gleichungslöser spielen bei der Lösung von großen linearen Gleichungssystemen eine entscheidende Rolle. Die klassischen iterativen Verfahren, wie das Jacobi Verfahren und das Gauss-Seidel Verfahren, basieren auf einer speziellen Aufteilung der Matrix $\mathbf{A}$, weswegen solche Verfahren auch Splitting-Methoden genannt werden. Durch die Zerlegung der Matrix $\mathbf{A}$ kann eine Iterationsvorschrift gewonnen werden, welche zur Lösung des Gleichungssystems dient. Jedoch ist die Konvergenzgeschwindigkeit dieser Verfahren oft nicht zufriedenstellend, d.h, bei großen Gleichungssystemen benötigen solche Verfahren sehr viele Iterationsschritte bis die Lösung erreicht wird. Um die Rechenzeit zu reduzieren werden heutzutage hauptsächlich moderne Iterationsverfahren eingesetzt. Eine der bekanntesten Verfahrensklassen für große schach-besetzte Matrizen sind die auf Projektionsmethoden basierenden Krylov-Unterraum Verfahren. Diese Verfahren zeichnen sich dadurch aus, dass nur Matrix-Vektor Multiplikationen und Skalarprodukte im Rechenablauf benötigt werden. Bei Verwendung spezieller Krylov-Unterraum Verfahren (z.B. CG, GMRES) konvergiert der Algorithmus spätestens nach n Iterationsschritten zur exakten Lösung, wobei n die Anzahl der Unbekannten darstellt. Aufgrund der Vielzahl dieser Verfahren werden im folgenden nur die zu dieser Arbeit relevanten Verfahren CG und BiCGStab (jeweils ohne und mit Vorkonditionierungstechniken) diskutiert. Zur weiteren ausführlicheren Erläuterung dieser Verfahrensklasse wird auf [2, 3, 4, 5] verwiesen.

1.3 Gliederung der Arbeit

Kapitel 2 stellt zunächst die Entstehung des linearen Gleichungssystems im Rahmen der FE-Methode dar. Anschießend werden auf die klassischen iterativen Verfahren (Jacobi und Gauss-Seidel Verfahren) eingegangen. Ihr Konvergenzverhalten wird anhand eines Modellproblems untersucht. Die bei den Fortran Implementationen verwendete Speichertechnik für große schwach-besetzte Matrizen wird zum Schluß dieses Kapitels eingeführt.

Kapitel 3 beschäftigt sich mit den modernen iterativen Gleichungslösern. Vorgestellt werden dabei das Verfahren der konjugierten Gradienten (CG-Verfahren) und das BiCGStab-Verfahren. Der Algorithmus der in den Verfahren häufig auftretenden Matrix-Vektor Multiplikation ist bezüglich der Speichertechnik gegeben. Ein wichtiges Thema in diesem Kapitel ist die Vorkonditionierung der iterativen Verfahren. Es werden verschiedene Möglichkeiten zur Vorkonditionierung diskutiert.

In Kapitel 4 wird die Assemblierung der zu lösenden linearen Gleichungssysteme im vorliegenden FE-Code beschrieben. Zunächst wird der Aufbau einiger notwendiger Hilfsgrößen und Hilfsvektoren vorgestellt. Danach wird für symmetrische und unsymmetrische Matrizen jeweils eine Version der Assemblierung der Koeffizientenmatrix (Steifigkeitsmatrix) gegeben.

In Kapitel 5 handelt es sich um die vergleichenden Studien zwischen den implementierten iterativen Gleichungslösern und dem vorliegeden Direktlöser (UMFPACK). Für ein ausgewählten Materialmodell der Festkörpermechanik werden die Lösungsverhalten der Verfahren bei unterschiedlichen Netzdichten (entspricht unterschiedlichen Größen des Gleichungssystems) untersucht.

Kapitel 6 fasst die Erfahrungen, die mit der vorgestellten Methode gemacht wurden, zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen und Verbesserungen.

Kapitel 2

Klassische Iterationsverfahren

In diesem Kapitel wird zunächst die Entstehung eines linearen Gleichungssystems aus Methode der Finiten-Elemente anhand eines Modellproblems erläutert. Anschießend werden die beiden klassischen Iterationsmethoden (Jacobi Verfahren und Gauss-Seidel Verfahren) zur Lösung des Gleichungssystems diskutiert. Zum Schluß wird die Implementation beider Verfahren für das Modellproblem gezeigt und das Konvergenzverhalten der Verfahren untersucht.

2.1 Entstehung des linearen Gleichungssystems

Die im Rahmen dieser Diplomarbeit betrachteten linearen Gleichungssysteme entstehen aus der Verwendung der Finite-Elemente-Methode, welches ein numerisches Verfahren zur näherungsweisen Lösung von Anfangsrandwertaufgaben ist. Hier wird nicht auf die konkrete Herleitung der FE-Methode eingegangen, sondern nur die Entstehung eines linearen Gleichungssystems am Beispiel eines Modellproblems gezeigt. Die entsprechende Theorie kann in den verschiedenen FEM-Lehrbüchern [7, 8, 9, 10] nachgelesen werden.

Als Beipiel wird ein einseitig eingespannter Zug-Druck-Stab betrachtet, der am freien Rand durch eine Zugkraft belastet wird (siehe Abb. 2.1). Die FE-Methode unterteilt den Stab in Teilgebiete. Hier wird zunächst eine äquidistante Unterteilung in drei Teilgebiete angenommen, wie in Abb. 2.2 dargestellt wird. Aus den linearen Ansatzfunktionen in den Teilgebieten ergibt sich die Elementsteifigkeitsmatrix im lokalen Koordinatensystem

$$\mathbf{k}^e = \frac{3EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad e = 1, 2, 3.$$

Die Systemfreiheitsgrade **r** (im globalen Koordinatensystem) können den Elementfreiheitsgraden **u** mit Hilfe der Inzidenzmatrix **Z** zugeordnet werden:

$$\mathbf{u} = \mathbf{Z}\mathbf{r} \quad \Longleftrightarrow \quad \begin{pmatrix} u_1^1 \\ u_2^1 \\ u_1^2 \\ u_2^2 \\ u_1^3 \\ u_2^3 \end{pmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}.$$

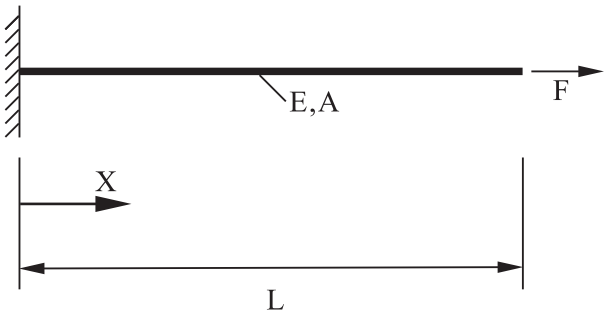


Abbildung 2.1: Zug-Druck-Stab unter Einzellast

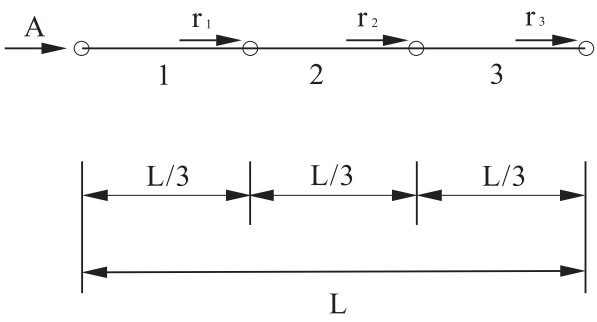


Abbildung 2.2: Unterteilung des Stabs

Die Zusammenfassung der Elementsteifigkeiten $\mathbf{k}^e$ führt auf die Systemsteifigkeitsmatrix **K**. Zu beachten ist, dass hierbei alle Elementsteifigkeiten auf das globale Koordinatensystem bezogen sein müssen. Dafür ist die Transformation der Elementsteifigkeiten $\mathbf{k}^e$ ins globale Koordinatensystem erforderlich. Die Beiträge der Elementsteifigkeiten zur Strukturtsteifigkeit werden mit $\mathbf{K}^e$ bezeichnet und es gilt für das Modellproblem

$$\mathbf{K}^1 = \frac{3EA}{L}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\mathbf{K}^2 = \frac{3EA}{L}\begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

und

$$\mathbf{K}^3 = \frac{3EA}{L}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}.$$

Mit dem Zusammenbau

$$\mathbf{K} = \sum_{e=1}^{3} \mathbf{K}^e$$

folgt die Systemsteifigkeit

$$\mathbf{K} = \frac{3EA}{L}\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}.$$

Zwischen dem Systemlastvektor $\mathbf{P} = (0,\, 0,\, \mathrm{F})^T$ und den Systemfreiheitsgraden **r** entsteht somit das lineare Gleichungssystem

$$\mathbf{Kr} = \mathbf{P} \quad \Longleftrightarrow \quad \frac{3EA}{L}\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}\begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ F \end{pmatrix}. \tag{2.1}$$

Bei einer äquidistaner Elementunterteilung hat das linare Gleichungssystem stets die Gestalt

$$\frac{nEA}{L}\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & & 0 \\ -1 & 2 & -1 & 0 & \cdots & & 0 \\ 0 & -1 & 2 & -1 & & & \vdots \\ \vdots & & & \ddots & & & \vdots \\ \vdots & & & & \ddots & & 0 \\ & & & & -1 & 2 & -1 \\ 0 & \cdots & & \cdots & 0 & -1 & 1 \end{bmatrix}\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ \vdots \\ \\ r_{n-1} \\ r_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ \vdots \\ \\ 0 \\ F \end{pmatrix},$$

wobei *n* die Anzahl der Teilgebiete (Elemente) angibt. Je feiner man das Gebiet des Stabs unterteilt, desto größer wird das Gleichungssystem. Aufgrund des linearen Ver-

schiebungsverlaufes in dem Stab ist die Lösung der Knotenverschiebungen bekannt:

$$\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ \vdots \\ \\ r_{n-1} \\ r_n \end{pmatrix} = \frac{L}{nEA} \begin{pmatrix} F \\ 2F \\ \vdots \\ \vdots \\ \\ (n-1)F \\ nF \end{pmatrix}$$

Es stehen zur Lösung des linearen Gleichungssystems viele Verfahren zur Verfügung, jedoch werden hier nur einige davon und deren Verhalten untersucht. Zunächst fangen wir mit den klassischen Iterationsverfahren an. Es ist noch zu beachten, dass die Systemsteifigkeitsmatrix für dieses Beispiel eine schwach-besetzte Bandmatrix mit der Bandbreite 3 ist. Um den Speicherplatz zu sparen werden deshalb bei der Umsetzung der Lösungsverfahren in Rechnerprogramme spezielle Speichertechniken verwendet, wobei nur die von Null verschiedenen Einträge im Rechner gespeichert werden.

2.2 Jacobi Verfahren

Das Jacobi Verfahren, auch als Gesamtschrittverfahren bezeichnet, ist ein elementarer Algorithmus der Numerischen Mathematik zur Näherungslösung von linearen Gleichungssystemen $\mathbf{Ax} = \mathbf{b}$. Es ist, wie das noch zu diskutierende Gauß-Seidel Verfahren, ein spezielles Splitting-Verfahren.

Wir nehmen an, dass das lineare Gleichungssystem die folgende Form besitzt:

$$\begin{array}{ccccccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1j}x_j & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2j}x_j & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{j1}x_1 & + & a_{j2}x_2 & + & \cdots & + & a_{jj}x_j & + & \cdots & + & a_{jn}x_n & = & b_j \\ \vdots & & \vdots & & & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nj}x_j & + & \cdots & + & a_{nn}x_n & = & b_n \end{array} \tag{2.2}$$

Die Idee des Jacobi Verfahrens beruht auf einer Zerlegung der Matrix $\mathbf{A}$ in die folgende Form

$$\mathbf{A} = \mathbf{D} + (\mathbf{A} - \mathbf{D}),$$

wobei $\mathbf{D} = diag\{a_{11}, \ldots, a_{nn}\}$ eine Diagonalmatrix ist und die Diagonalelemente von $\mathbf{A}$ umfasst. Mit der Voraussetzung $a_{ii} \neq 0$ $(i = 1, \ldots, n)$ kann das Gleichungssystem $\mathbf{Ax} =$

b somit in folgende Form umgeschrieben werden:

$$\mathbf{D}\mathbf{x} = (\mathbf{D} - \mathbf{A})\mathbf{x} + \mathbf{b}.$$

Dadurch wird eine Iterationsvorschrift für den gesuchten Vektor **x** gewonnen:

$$\mathbf{x}_{m+1} = \mathbf{D}^{-1}(\mathbf{D} - \mathbf{A})\mathbf{x}_m + \mathbf{D}^{-1}\mathbf{b}. \tag{2.3}$$

Für die Durchführung der Iteration muss man hier zuerst einen beliebigen Startvektor $\mathbf{x}_0$ festlegen. Da es der erste Iterationsschritt ist, hat m dabei den Wert Null. Das Ergebnis der Rechnung liefert einen ersten Näherungswert für den gesuchten Lösungsvektor. Diesen Näherungswert kann man wiederum in die Iterationsvorschrift einsetzen und gewinnt einen besseren Näherungswert, den man wieder einsetzen kann. Wiederholt man diesen Vorgang, gewinnt man eine Folge von Werten, die sich dem Lösungsvektor immer mehr annähern, wenn die speziellen Konvergenzbedingungen (siehe [2], S.63) erfüllt sind:

$$\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots, \longrightarrow \mathbf{x}.$$

Betrachtet man nun das Ergebnis nach dem m-ten Iterationsschritt

$$\mathbf{x}_m = (x_{m,1}, x_{m,2}, \ldots, x_{m,j}, \ldots, x_{m,n})^T,$$

somit kann man entsprechend das Ergebnis nach der $(m+1)$-te Iteration wie

$$\mathbf{x}_{m+1} = (x_{m+1,1}, x_{m+1,2}, \ldots, x_{m+1,j}, \ldots, x_{m+1,n})^T$$

darstellen. Die Komponentenschreibweise von Gl.(2.3) lautet dann

$$\underbrace{\begin{pmatrix} x_{m+1,1} \\ \vdots \\ \vdots \\ x_{m+1,n} \end{pmatrix}}_{\mathbf{x}_{m+1}} = \underbrace{\begin{bmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{a_{nn}} \end{bmatrix}}_{\mathbf{D}^{-1}} \underbrace{\begin{bmatrix} 0 & -a_{12} & \cdots & -a_{1n} \\ -a_{21} & \ddots & & -a_{2n} \\ \vdots & & \ddots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & 0 \end{bmatrix}}_{\mathbf{D}-\mathbf{A}} \underbrace{\begin{pmatrix} x_{m,1} \\ \vdots \\ \vdots \\ x_{m,n} \end{pmatrix}}_{\mathbf{x}_m} + \cdots$$

$$\cdots + \underbrace{\begin{bmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{a_{nn}} \end{bmatrix}}_{\mathbf{D}^{-1}} \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}}_{\mathbf{b}}$$

oder alternativ

$$x_{m+1,i} = \frac{1}{a_{ii}} \left(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_{m,j} + b_i \right) \qquad i = 1, 2, \ldots, n.$$

Die Iterationsformel benutzt die i-te Zeile der Matrix $\mathbf{A}$ für die Lösung von $x_{m+1,i}$ durch eine lineare Kombination der Werte $x_{m,i}$, $i = 1, \ldots, n$, des vorherigen Iterationsschrittes m:

$$x_{m+1,i} = \frac{b_i - a_{i1} x_{m,1} - \ldots - a_{ii-1} x_{m,i-1} - a_{ii+1} x_{m,i+1} - \ldots - a_{in} x_{m,n}}{a_{ii}},$$

für $i = 1, 2, \ldots, n$. Es ist somit offensichtlich, dass das Jacobi Verfahren nur die alten Werte für den neuen Iterationsschritt benötigt.

Die Konvergenz des Jacobi Verfahrens wird wie bei allen Splitting-Verfahren mittels des Banachschen Fixpunktsatzes (siehe [2], S. 30) untersucht. Das Verfahren konvergiert genau dann, wenn der Spektralradius[1] der Iterationsmatrix $\mathbf{D}^{-1}(\mathbf{D} - \mathbf{A})$ kleiner als eins ist. Insbesondere ergibt sich dies, wenn die Systemmatrix $\mathbf{A}$ strikt diagonaldominant ist, d.h., es gilt stets

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|$$

für $i = 1, 2, \ldots, n$. Oft kann die Konvergenz auch unter einer abgeschwächten Forderung bewiesen werden, wobei in dem in [2], S. 64 vorliegenden Satz die Irreduzibilität der Matrix $\mathbf{A}$ gefordert wird.

2.3 Gauss-Seidel Verfahren

Ähnlich wie das Jacobi Verfahren ist das Gauss-Seidel Verfahren (Einzelschrittverfahren) ebenfalls ein iteratives Verfahren zur Näherungslösung des linearen Gleichungssystems (1.1).

Als Beispiel wird jetzt das im vorherigen Abschnitt erwähnte lineare Gleichungssystem (2.2) nochmals betrachtet. Das Gauss-Seidel Verfahren kann ebenfalls über eine Zerlegung der Matrix $\mathbf{A}$ eingeführt werden

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U},$$

[1] Betragsmäßig größter Eigenwert einer Matrix.

wobei **D** die Diagonalmatrix, **L** eine strikte untere Dreiecksmatrix und **U** eine strikte obere Dreiecksmatrix darstellt. Nach der Umstellung des Gleichungssystems

$$(\mathbf{L}+\mathbf{D})\mathbf{x} = -\mathbf{U}\mathbf{x} + \mathbf{b}$$

ergibt sich wiederum eine Iterationsvorschrift für das Gauss-Seidel Verfahren:

$$\mathbf{x}_{m+1} = -(\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}\mathbf{x}_m + (\mathbf{L}+\mathbf{D})^{-1}\mathbf{b}. \tag{2.4}$$

Wird Gl.(2.4) in der folgenden Form

$$(\mathbf{L}+\mathbf{D})\mathbf{x}_{m+1} = -\mathbf{U}\mathbf{x}_m + \mathbf{b}$$

umgeschrieben, so kann sie leicht in Komponentenschreibweise dargestellt werden:

$$\sum_{j=1}^{i} a_{ij}x_{m+1,i} = \frac{1}{a_{ii}}\left(-\sum_{j=i+1}^{n} a_{ij}x_{m,j} + b_i\right) \qquad i,j = 1,2,\cdots,n$$

Für die i-te Komponente von $\mathbf{x}_{m+1}$ gilt dann ähnlich wie beim Jacobi Verfahren die folgende Formel:

$$x_{m+1,i} = \frac{b_i - a_{i1}x_{m+1,1} - \ldots - a_{ii-1}x_{m+1,i-1} - a_{ii+1}x_{m,i+1} - \ldots - a_{in}x_{m,n}}{a_{ii}}.$$

Es ist dabei leicht zu erkennen, dass die bereits berechneten Werte des aktuellen Iterationsschritts mit verwendet werden. Diese Vorgehensweise bringt gegenüber dem Jacobi Verfahren zwei wesentliche Vorteile:

- Das Gauss-Seidel Verfahren konvergiert normalerweise schneller als das Jacobi Verfahren und
- Es wird kein zusätzlicher Speicherplatz im neuen Iterationsschritt für den Vektor $\mathbf{x}_{m+1}$ benötigt, weil jede Komponente von $\mathbf{x}_m$ sofort mit dem neuen Wert aktualisiert werden kann. Diese Ersetzung wird, ausgehend von einer willkürlichen Startbelegung der Variablen, sukzessive wiederholt. Bei dem Jacobi Verfahren muss jedoch nach einem kompletten Iterationsschritt das Kopieren eines Vektors erfolgen, wodurch ein zweiter Vektor abgespeichert werden muss. Dies wäre bei großen linearen Gleichungssystemen nicht erwünscht.

Es kann nachgewiesen werden, dass das Gauss-Seidel Verfahren für eine strikt diagonaldominante Matrix **A** konvergiert (siehe [2], S. 70). Wie wir im nächsten Abschnitt sehen, ist die Konvergenzgeschwindigkeit des Gauss-Seidel Verfahrens meistens höher als die des Jacobi Verfahrens.

2.4 Implementation der Verfahren

In diesem Abschnitt wird die Implementation beider Verfahren sowie die Konvergenzverhalten am Beispiel von dem im Abschnitt 2.1 erwähnten Modellproblem jedoch für $n = 2$ gezeigt. Hierzu soll beachtet werden, dass die Implementation an das spezielle Speicherformat der Matrix **A** angepasst werden muss.

2.4.1 Speichertechnik für große schwach-besetzte Matrizen

Für große schwach-besetzte Matrizen wird praktisch im Rechner nicht so viel Speicherplatz entsprechend ihren Dimensionen reserviert, d.h., es wird für die Matrix **A** kein 2-dimensionales Feld (Dimension $n \times n$) definiert. Bei der Lösung des Gleichungssystems sind nur solche Elemente der Matrix **A** von Interesse, die von Null verschieden sein können. Dazu kann man im Rechner drei Vektoren von der Länge $amount$ definieren, wobei $amount$ die Anzahl der von Null verschiedenen Elemente der Matrix **A** angibt. Im ersten Vektor **v** werden die Werte dieser Elemente registriert. Ihre Lagen in der Matrix **A** werden in den anderen beiden Vektoren gespeichert, nämlich die Spaltennummern in einem Vektor **c** und die Zeilennummern in dem Vektor **r**. Diese Vorgehensweise zur Speicherung der Elemente der Matrix **A** ist für große schwach-besetzte Gleichungssysteme besonders sinnvoll. Einerseits wird der Speicherplatzbedarf wegen der Schwachbesetztheit stark verringert und andererseits kann man viele unnötigen Nulloperationen (Rechenoperationen mit Null) vermeiden. Für das im Abschnitt 2.1 diskutierten Beispiel sieht die Speicherbelegung der Matrix **A** wie folgt aus,

v :	2.	-1.	-1.	2.	-1.	$\cdots$
r :	1	1	2	2	2	$\cdots$
c :	1	2	1	2	3	$\cdots$

wobei hier eine zeilenweise Abspeicherung verwendet wird.

2.4.2 Implementation und Konvergenzverhalten

Aufgrund der soeben erläuterten Speichertechnik werden bei der Implementation die rechte Seite des Gleichungssystems **b** sowie die drei Vektoren **v**, **r**, **c** anstatt der Matrix **A** an die entsprechenden Unterroutinen übergeben.

Der Algorithmus des Jacobi Verfahrens ist in Tabelle 2.1 beschrieben. Dabei wird der Nullvektor als Startvektor gewählt. Beim Gauss-Seidel Verfahren sieht der Algorithmus ähnlich aus, welcher in Tabelle 2.2 dargestellt ist. In den Anhängen A.1 und A.2 werden die implementierten Fortran-Routinen des Jacobi und des Gauss-Seidel Verfahrens

angegeben.

Für das Modellproblem liefert das Jacobi Verfahren bei einer Diskretisierung in zwei Elemente (und damit insgesamt 2 Unbekannte im Gleichungssystem) mit

$$\mathbf{A} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \quad \text{und} \quad \mathbf{b} = \begin{bmatrix} 0 \\ \frac{2}{3} \cdot 10^{-2} \end{bmatrix}$$

den folgenden Konvergenzverlauf:

Jacobi Verfahren				
k	$x_{k,1}$	$x_{k,2}$	$\varepsilon_k = \|\mathbf{x_k} - \mathbf{A^{-1}b}\|$	$\varepsilon_k/\varepsilon_{k-1}$
0	0	0	1.333e-02	
15	6.614583333333e-03	1.328125000000e-02	5.208e-05	5.000e-01
30	6.666463216146e-03	1.333292643229e-02	4.069e-07	1.000e+00
45	6.666665077209e-03	1.333333174388e-02	1.590e-09	5.000e-01
60	6.666666660458e-03	1.333333332092e-02	1.242e-11	1.000e+01
75	6.666666666618e-03	1.333333333328e-02	4.851e-14	5.000e-01
90	6.666666666666e-03	1.333333333333e-02	3.799e-16	1.000e+01
92	6.666666666667e-03	1.333333333333e-02	1.891e-16	9.954e-01

Hierbei wird das Konvergenzkriterium nach 92 Iterationsschritten erst erfüllt. Dementsprechend sieht der Konvergenzverlauf des Gauss-Seidel Verfahrens folgendermaßen aus:

Gauss-Seidel Verfahren				
k	$x_{k,1}$	$x_{k,2}$	$\varepsilon_k = \|\mathbf{x_k} - \mathbf{A^{-1}b}\|$	$\varepsilon_k/\varepsilon_{k-1}$
0	0	0	1.333e-02	
15	6.666259765625e-03	1.333292643229e-02	4.069e-07	5.000e-01
30	6.666666654249e-03	1.333333332092e-02	1.242e-11	5.000e-01
45	6.666666666666e-03	1.333333333333e-02	3.799e-16	5.011e-01
48	6.666666666667e-03	1.333333333333e-02	4.770e-17	5.000e-01

Wie Abb. 2.3 zeigt, konvergiert das Gauss-Seidel Verfahren unter gleichen Bedingungen fast doppelt so schnell wie das Jacobi Verfahren. Aber trotzdem ist die Konvergenzrate nicht zufriedenstellend, weil allein bei einem kleinen Gleichungssystem mit nur 2 Unbekannten schon fast 50 Iterationsschritte für die Näherungslösung benötigt wird. Mit so hohem Aufwand werden die Verfahren bei großen Gleichungssystemen schon nicht mehr praktikabel. Im nächsten Kapitel wird auf die zur Lösung von großen linearen Gleichungssystemen sehr effizienten Verfahren, die sogenannten Krylov-Unterraum Verfahren, eingegangen und die wichtigsten Varianten dieser Verfahren im Rahmen dieser Arbeit diskutiert.

Wähle tol, $maxit$

$k = 1$, $conv = 1$

Für $i = 1, \ldots, n$

- $x(i) = 0$
- $xnew(i) = 0$

Solange ($k < maxit$) und ($conv > tol$)

- $j = 1$
- Für $i = 1, \ldots, n$
  - $sum = b(i)$
  - Solange $j \leq amount$
    - Y: $r(j) = i$ — N:
    - Y: $r(j) \neq c(j)$ — N: Exit
    - $sum = sum - v(j) * x(c(j))$ | $temp = temp + v(j)$
    - $j = j + 1$
  - $xnew(i) = sum/temp$
- $conv = 0$
- Für $i = 1, \ldots, n$
  - $conv = conv + |xnew(i) - x(i)|$
  - $x(i) = xnew(i)$
- $k = k + 1$

Y: $conv \leq tol$ — N:

Print „Jacobi method successful“	Print „Maximal iteration step reached“

Tabelle 2.1: Algorithmus des Jacobi Verfahrens

Wähle tol, $maxit$
$k = 1$, $conv = 1$
Für $i = 1, \ldots, n$
$x(i) = 0$
$xold(i) = 0$
Solange ($k < maxit$) und ($conv > tol$)
$j = 1$
Für $i = 1, \ldots, n$
$sum = b(i)$
Solange $j \leq amount$
Y: $r(j) = i$ — N: Exit
Y: $r(j) \neq c(j)$ — N: Exit
$sum = sum - v(j) * x(c(j))$ | $temp = temp + v(j)$
$j = j + 1$
$x(i) = sum/temp$
$conv = 0$
Für $i = 1, \ldots, n$
$conv = conv + \|x(i) - xold(i)\|$
$xold(i) = x(i)$
$k = k + 1$
Y: $conv \leq tol$ — N
Print „Gauss-Seidel method successful“ | Print „Maximal iteration step reached“

Tabelle 2.2: Algorithmus des Gauss-Seidel Verfahrens

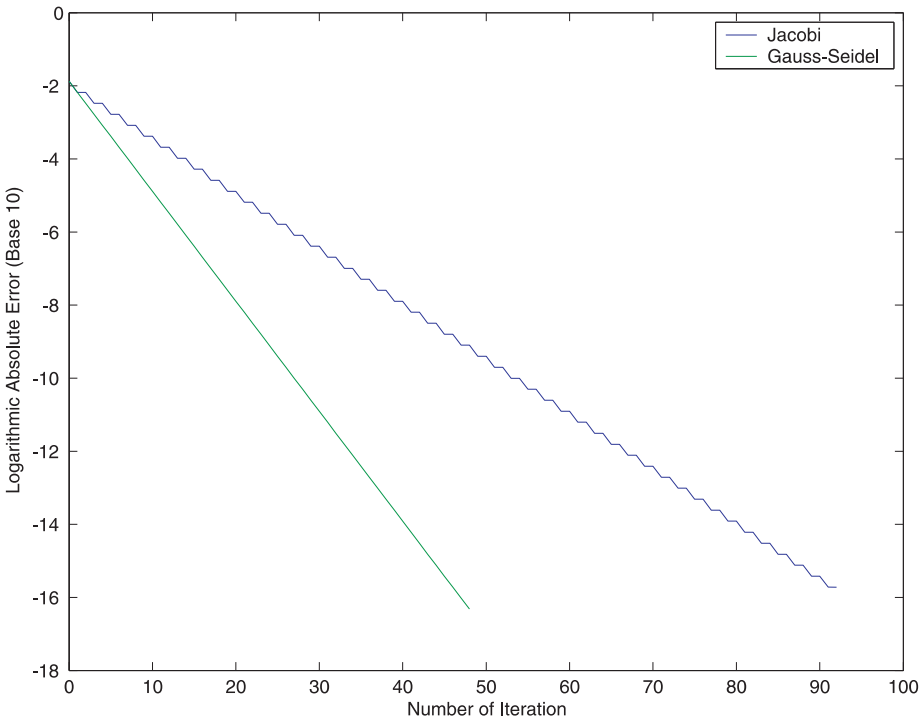


Abbildung 2.3: Konvergenzverhalten der Jacobi und Gauss-Seidel Verfahren am Beispiel des Zug-Druck-Stabs unter Einzellast ($n = 2$)

Kapitel 3

Moderne Iterationsverfahren

Es sind in diesem Kapitel einige Varianten der Krylov-Unterraum Verfahren zu diskutieren, welche effiziente iterative Lösungsverfahren für große lineare Gleichungssysteme darstellen. Zunächst wird eine kurze Einführung gegeben, wobei die Grundgedanken und die theoretischen Grundlagen der Krylov-Unterraum Verfahren erläutert werden. Im Anschluß wird das Verfahren der konjugierten Gradienten (das CG-Verfahren) sowie das BiCGStab-Verfahren im Detail präsentiert und zum Schluß die auf die beiden Verfahren bezogenen Vorkonditionierer eingeführt, die zur Verbesserung des Konvergenzverhaltens dienen.

3.1 Einführung

Krylov-Unterraum Verfahren sind iterative Verfahren zum Lösen von großen, schwach-besetzten linearen Gleichungssystemen, wie sie zum Beispiel mit Methode der finiten Elemente bei der Diskretisierung von partiellen Differentialgleichungen entstehen. Sie gehören derzeit zu den wichtigsten und effizientesten iterativen Gleichungslösern für große lineare Gleichungssysteme. Es existieren eine Reihe von iterativen Gleichungslösern im Rahmen der Krylov-Unterraum Verfahren.

Wie bei den Splitting-Methoden werden bei den Krylov-Unterraum Verfahren ebenfalls nur Matrix-Vektor Multiplikationen und Skalarprodukte im Rechenablauf benötigt. Die Matrix-Vektor Multiplikation kostet bei einer schwach-besetzten Matrix mit $O(n)$ Einträgen nur $O(n)$ arithmetische Operationen. Damit liegt der Gesamtaufwand bei $m \ll n$ Iterationen immer noch bei $O(n)$, allerdings mit einem sehr hohen Vorfaktor. Deswegen sind Krylov-Unterraum Verfahren für lineare Gleichungssysteme mit schwach-besetzten Koeffizientenmatrizen höherer Ordnung geeignet.

Wir betrachten erneut das lineare Gleichungssystem $\mathbf{Ax} = \mathbf{b}$. Mit einer beliebigen Näherungslösung $\mathbf{x}_0$ für $\mathbf{x}$ bilden wir das Residuum $\mathbf{r}_0 = \mathbf{b} - \mathbf{Ax}_0$. Der zugehörige m-te Krylov-Unterraum $\mathcal{K}_m$ ist dann der von den Vektoren $\mathbf{r}_0$, $\mathbf{Ar}_0$, ..., $\mathbf{A}^{m-1}\mathbf{r}_0$ aufgespannte Untervektorraum (siehe [3], S. 152):

$$\mathcal{K}_m(\mathbf{A}, \mathbf{r}_0) = span\{\mathbf{r}_0, \mathbf{Ar}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{m-1}\mathbf{r}_0\}.$$

Fast alle Verfahren finden nun eine Näherungslösung $\mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m$ mit der Bedingung, dass der Vektor $\mathbf{b} - \mathbf{Ax}_m$ orthogonal zu allen Vektoren eines Unterraumes $\mathcal{L}_m$ steht, eine sogenannte Projektion:

$$\mathbf{b} - \mathbf{Ax}_m \perp \mathcal{L}_m \tag{3.1}$$

Für den Fall $\mathcal{L}_m = \mathcal{K}_m$, d.h., der Residuenvektor $\mathbf{r}_m = \mathbf{b} - \mathbf{Ax}_m$ steht senkrecht auf $\mathcal{K}_m$, liegt ein orthogonales Projektionsverfahren vor. Diese Bedingung heißt Galerkin Bedingung. Wenn die Beziehung $\mathcal{L}_m \neq \mathcal{K}_m$ gilt, liegt eine schiefe Projektionsmethode vor und (3.1) wird als Petrov-Galerkin Bedingung bezeichnet (siehe [2], S. 106).

Damit ist das Problem auf ein m-dimensionales lineares Gleichungssystem reduziert. Das Ganze wird zu einem iterativen Lösungsverfahren, wenn man die Dimension in jedem Schritt um eins erhöht.

Aus Sicht der Approximationstheorie besitzen die mit den Krylov-Unterraum Verfahren erhaltenen Näherungslösungen die Form

$$\mathbf{A}^{-1}\mathbf{b} \approx \mathbf{x}_m = \mathbf{x}_0 + q^{m-1}(\mathbf{A})\mathbf{r}_0,$$

wobei q^{m-1} ein bestimmtes Polynom vom Grad $m-1$ ist. Für den einfachen Fall $\mathbf{x}_0 = \mathbf{0}$ erhalten wir

$$\mathbf{A}^{-1}\mathbf{b} \approx q^{m-1}(\mathbf{A})\mathbf{b},$$

wobei $\mathbf{A}^{-1}\mathbf{b}$ nämlich von $q^{m-1}(\mathbf{A})\mathbf{b}$ approximiert wird.

Die unterschiedlichen Versionen der Krylov-Unterraum Verfahren ergeben sich durch die konkrete Wahl des Raumes $\mathcal{L}_m$, sowie durch Ausnutzen von speziellen Eigenschaften der Matrix $\mathbf{A}$, was das Verfahren beschleunigt, aber die Anwendbarkeit auch einschränkt. Die einfachste Variante ist, für $\mathcal{L}_m$ einfach wieder den Krylov-Unterraum selbst zu wählen. Ist die Matrix schwach-besetzt, so ist das Matrix-Vektor-Produkt schnell ausrechenbar und der Algorithmus praktikabel. Ein Beispiel ist das Verfahren der konjugierten Gradienten (CG-Verfahren), auf das im nächsten Abschnitt eingegangen wird. Hierbei ist $\mathcal{L}_m = \mathcal{K}_m$ und es ist für symmetrische, positiv definite Matrizen gedacht. Eine andere Variante basiert auf die Definition von $\mathcal{L}_m$ als ein Krylov-Unterraum Verfahren assoziiert mit $\mathbf{A}^T$, nämlich $\mathcal{L}_m = \mathcal{K}_m^T = span\{\mathbf{r}_0, \mathbf{A}^T\mathbf{r}_0, \ldots, (\mathbf{A}^T)^{m-1}\mathbf{r}_0\}$. Eine Methode dieser Variante wird in diesem Kapitel diskutiert, nämlich das BiCGStab-Verfahren, welches geeignet für unsymmetrische Matrizen ist.

Man erhält somit eine Klasse an Verfahren. Es gibt jedoch kein deutlich herausragendes

allgemein anwendbares Verfahren. Daher muss für jede Problemklasse von Gleichungssystemen ein geeignetes Verfahren gefunden werden. Viel wichtiger als die Auswahl der speziellen Krylov-Unterraum Verfahren ist die Wahl des Vorkonditionierers. Dieser formt das lineare Gleichungssystem äquivalent um, so dass die Lösung unverändert bleibt, sich aber günstigere Eigenschaften für die Konvergenz ergeben. Hier sind entscheidende Geschwindigkeitsgewinne zu erzielen, die dazu führen, dass selbst Systeme mit Millionen Unbekannten in wenigen Dutzend Schritten zufriedenstellend gelöst werden können. Bezogen auf das CG-Verfahren und das BiCGStab-Verfahren werden zum Schluß dieses Kapitels einige Vorkonditionierer vorgestellt.

3.2 Das CG-Verfahren

Das CG-Verfahren (Verfahren der konjugierten Gradienten) ist eines der bekanntesten iterativen Verfahren zum Lösen von großen linearen Gleichungssystemen mit schwachbesetzten, symmetrischen und positv definiten Koeffizientenmatrizen. Dieses Verfahren realisiert eine orthogonale Projektion auf den Krylov-Unterraum $\mathcal{K}_m(\mathbf{r}_0, \mathbf{A})$, wobei $\mathbf{r}_0$ das Anfangsresiduum ist. Die Idee hinter dem CG-Verfahren beruht auf der Verwendung von „Korrekturrichtungen“ bei der Aufdatierung der Iteration.

Im folgenden wird angenommen, dass die Matrix $\mathbf{A}$ symmetrisch und positiv definit ist, $\mathbf{A} = \mathbf{A}^T$, $\mathbf{v}^T\mathbf{A}\mathbf{v} > 0$ für $\mathbf{v} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$. Dann kann man dem Gleichungssystem die Funktion[1]

$$f(\mathbf{x}) := \frac{1}{2} < \mathbf{A}\mathbf{x}, \mathbf{x} > - < \mathbf{b}, \mathbf{x} >$$

zuordnen. Der Gradient dieser Funktion

$$\mathbf{r} := -grad\{f(\mathbf{x})\} = -\frac{\partial f}{\partial \mathbf{x}} = \mathbf{b} - \mathbf{A}\mathbf{x} \tag{3.2}$$

ist dann ebenfalls ein Defekt, nämlich das Residuum $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ des linearen Gleichungssystems. Ist dieser Defekt gleich Null, so nimmt die Funktion $f(\mathbf{x})$ wegen positiver Definitheit von $\mathbf{A}$ ein Minimum an. Findet man nun ein $\mathbf{x}$, für das die Funktion $f(\mathbf{x})$ ein Minimum hat, dann ist auch das Gleichungssystem (1.1) erfüllt. Die Idee bei den Verfahren der konjugierten Gradienten besteht nun darin, dass man $f(\mathbf{x})$ hinsichtlich einer bestimmten (eindimensionalen) Suchrichtung $\mathbf{p}$ minimiert. Die Wahl von $\mathbf{p}$ hat demzufolge einen entscheidenden Einfluß auf die Qualität der Iteration. Als Ansatz könnte man die Richtung des steilsten Abstiegs bzw. des negativen Gradienten, also das Residuum (3.2), verwenden. Diese als Gradientenverfahren bezeichnete Methode weist zumeist ein schlechtes Konvergenzverhalten auf, weil sich die nach einer Iteration gewonnene Verbesserung von $\mathbf{x}$ in die Richtung $\mathbf{p}_m$ unter Verwendung der Energienorm bei der folgenden Iteration in Richtung $\mathbf{p}_{m+1}$ wieder bezüglich $\mathbf{p}_m$ verschlechtern kann. Um

[1] $< \mathbf{a}, \mathbf{b} >$ stellt das Skalarprodukt der Vektor $\mathbf{a}$ und $\mathbf{b}$ dar, $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$.

dies zu verhindern muss eine Suchrichtung gefunden werden, die konjugiert zu (allen) vorhergegangenen Suchrichtungen ist. Man fordert also, dass $\mathbf{x}_{m+1} = \mathbf{x}_m + \mathbf{q}$ bezüglich $\mathbf{p}_m$ optimal bleibt, wenn $\mathbf{x}_m$ optimal bezüglich $\mathbf{p}_m$ ist. Dies ist erfüllt, wenn die beiden Vektoren $\mathbf{q}$ und $\mathbf{p}_m$ konjugiert sind, $\mathbf{p}_m^T \mathbf{A}\mathbf{q} = 0$, und somit $\mathbf{A}\mathbf{q} \perp \mathbf{p}_m$ gilt.

Diese kurze Beschreibung des konjugierten Gradientenverfahrens soll einen Einblick in den grundsätzlichen Aufbau solcher Verfahren geben. Da der Krylov-Raum $\mathcal{K}_n$ mit dem $\mathbb{R}^n$ übereinstimmt, liefert das CG-Verfahren spätestens nach n Schritten die exakte Lösung des Systems (1.1) bei Verwendung einer exakten Arithmetik. Für große Systeme wird man das Verfahren jedoch in der Regel eher abbrechen. In diesem Zusammenhang ist man daher an einer hohen Konvergenzgeschwindigkeit interessiert. In Tabelle 3.1 wird der Algorithmus des CG-Verfahrens dargestellt.

Wähle $\mathbf{x}_0 \in \mathbb{R}^n$		
$\mathbf{p}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\alpha_0 := \|\mathbf{r}_0\|_2^2$		
Für $m = 0, \ldots, n-1$		
Y	$\alpha_m \neq 0$	N
$\mathbf{v}_m := \mathbf{A}\mathbf{p}_m, \lambda_m := \dfrac{\alpha_m}{< \mathbf{v}_m, \mathbf{p}_m >}$		Exit
$\mathbf{x}_{m+1} := \mathbf{x}_m + \lambda_m \mathbf{p}_m$		
$\mathbf{r}_{m+1} := \mathbf{r}_m - \lambda_m \mathbf{v}_m$		
$\alpha_{m+1} := \|\mathbf{r}_{m+1}\|_2^2$		
$\mathbf{p}_{m+1} := \mathbf{r}_{m+1} + \dfrac{\alpha_{m+1}}{\alpha_m}\mathbf{p}_m$		

Tabelle 3.1: Algorithmus des CG-Verfahrens

3.3 Das BiCGStab-Verfahren

Ein weiteres Verfahren, das zur Lösung unsysmmetrischer Systeme in den letzten Jahren vermehrt eingesetzt wurde, ist das von van der Vorst [11] vorgeschlagene BiCGStab Verfahren (BiCG Stabilized), das eine stabilisierte Version des bikonjugierten Gradientenverfahrens (siehe [2] S. 158ff) darstellt. Da sich dieses Verfahren als Kombination aus

mehreren hier nicht diskutierten iterativen Verfahren ergibt und relativ kompliziert ist, wird die ausführliche Erläuterung auf [3], S. 231ff verwiesen.

Wähle $\mathbf{x}_0 \in \mathbb{R}^n$ und $\varepsilon, \tilde{\varepsilon} > 0$		
$\mathbf{p}_0 := \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\rho_0 :=< \mathbf{r}_0, \mathbf{r}_0 >$, $j := 0$		
Solange $\|\mathbf{r}_j\|_2 > \varepsilon$		
$\mathbf{v}_j := \mathbf{A}\mathbf{p}_j$		
$\sigma_j :=< \mathbf{v}_j, \mathbf{r}_0 >$		
Y $\sigma_j > \tilde{\varepsilon}\|\mathbf{v}_j\|_2\|\mathbf{r}_0\|_2$ N		
$\alpha_m := \frac{\rho_j}{\sigma_j}$		Restart mit $\mathbf{x}_0 = \mathbf{x}_j$
$\mathbf{s}_j := \mathbf{r}_j - \alpha_j \mathbf{v}_j$		
Y $\|\mathbf{s}_j\|_2 > \varepsilon$ N		
$\mathbf{t}_j := \mathbf{A}\mathbf{s}_j$	$\mathbf{x}_{j+1} := \mathbf{x}_j + \alpha_j \mathbf{p}_j$	
$\omega_j := \frac{< \mathbf{t}_j, \mathbf{s}_j >}{< \mathbf{t}_j, \mathbf{t}_j >}$	$\mathbf{r}_{j+1} := \mathbf{s}_j$	
$\mathbf{x}_{j+1} := \mathbf{x}_j + \alpha_j \mathbf{p}_j + \omega_j \mathbf{s}_j$	$j := j + 1$	
$\mathbf{r}_{j+1} := \mathbf{s}_j - \omega_j \mathbf{t}_j$		
$\rho_{j+1} :=< \mathbf{r}_{j+1}, \mathbf{r}_0 >$		
$\beta_j := \frac{\alpha_j}{\omega_j} \frac{\rho_{j+1}}{\rho_j}$		
$\mathbf{p}_{j+1} := \mathbf{r}_{j+1} + \beta_j(\mathbf{p}_j - \omega_j \mathbf{v}_j)$		
$j = j + 1$		

Tabelle 3.2: Algorithmus des BiCGStab-Verfahrens

Wie das BiCG-Verfahren stellt das BiCGStab-Verfahren ebenfalls ein Krylov-Unterraum Verfahren dar und die Orthogonalität ist gegeben durch die Petrov-Galerkin Bedingung

$$\mathcal{L}_m = \mathcal{K}_m^T(\mathbf{A}^T, \mathbf{r}_0) = span\{\mathbf{r}_0, \mathbf{A}^T\mathbf{r}_0, \ldots, (\mathbf{A}^T)^{m-1}\mathbf{r}_0\}.$$

Im Vergleich mit dem BiCG-Verfahren weist das BiCGStab-Verfahren ein besseres Konvergenzverhalten auf. Jedoch könnte beim BiCGStab-Verfahren ein frühzeitiger Abbruch der Iteration auftreten, ohne das richtige Ergebnis berechnet wird (siehe [2], S. 172). Dort wird eine stabilisierte Version des BiCGStab-Verfahrens vorgeschlagen. In Tabelle 3.2 wird der Algorithmus dieses verbesserten BiCGStab-Verfahrens gezeigt.

3.4 Matrix-Vektor Multiplikation

Im Algorithmus des CG- und des BiCGStab-Verfahrens treten Matrix-Vektor Multiplikationen auf. Ein geschickter Algorithmus zur Durchführung dieser Operation kann die gesamte Rechenzeit positv beeinflussen, da sie in jedem Iterationsschritt mehrfach vorkommt.

Dieser Algorithmus sollte an das spezielle Speicherformat der Matrix **A** angepasst werden. Dazu ist eine zusätzliche Unterroutine erforderlich. Als Eingangsgrößen werden die Vektoren **v**, **r** und **c** sowie die Skalaren n und $amount$ an die Unterroutine übergeben, deren Bedeutungen im Abschnitt 2.4.1 zu finden sind. Ferner steht **vec** für den mit der Matrix zu multiplizierenden Vektor, der ebenfalls eine Eingangsgröße ist. Der Vektor **res** ist das Ergebnis der Matrix-Vektor Multiplikation.

In Tabelle 3.3 wird der Algorithmus dieser Unterroutine dargestellt. Hier erkennt man, dass es eine einfache Schleife über den Vektor **v** genügt. Mit Hilfe der Vektoren **c** und **r** werden die Werte von **v** mit den entprechenden Elementen von **vec** miltipliziert und dann zu den zugehörigen Elementen von **res** addiert.

Für $i = 1, \ldots, n$
$res(i) = 0$
Für $i = 1, \ldots, amount$
$res(r(i)) = res(r(i)) + v(i) * vec(c(i))$

Tabelle 3.3: Algorithmus der Matrix-Vektor Multiplikation

3.5 Vorkonditionierungstechniken

Neben den verschiedenen Lösungsverfahren spielt die Vorkonditionierung ebenfalls eine große Rolle beim Lösen des linearen Gleichungssystems. Die Vorkonditionierung ist eine Technik, bei der das lineare Gleichungssystem $\mathbf{Ax} = \mathbf{b}$ äquivalent umformuliert wird. Bei Verwendung der Vorkonditionierung bleibt die Lösung des Gleichungssystems erhalten. Zielsetzung der Vorkonditionierung ist die Verringerung der Konditionszahl der Koeffizientenmatrix des Gleichungssystems, da das Konvergenzverhalten wesentlich von der Kondition der Koeffizientenmatrix abhängt.

Die Idee der Vorkonditionierung besteht darin, das lineare Gleichungssystem $\mathbf{Ax} = \mathbf{b}$ durch ein äquivalentes System

$$\begin{aligned} \mathbf{P}_L\mathbf{A}\mathbf{P}_R\mathbf{y} &= \mathbf{P}_L\mathbf{b}, \\ \mathbf{x} &= \mathbf{P}_R\mathbf{y} \end{aligned}$$

zu ersetzen, wobei die regulären Matrizen $\mathbf{P}_L$ und $\mathbf{P}_R$ jeweils als Linksvorkonditionierer und Rechtsvorkonditionierer bezeichnet werden. Ist $\mathbf{P}_L \neq \mathbf{I}$ und $\mathbf{P}_R = \mathbf{I}$, so heißt das System linksvorkonditioniert. Die Rechtsvorkonditionierung gilt dementsprechend, wenn $\mathbf{P}_R \neq \mathbf{I}$ und $\mathbf{P}_L = \mathbf{I}$. Des Weiteren heißt das System beiderseitig vorkonditioniert, wenn $\mathbf{P}_L \neq \mathbf{I} \neq \mathbf{P}_R$. Diese Art der Vorkonditionierung findet insbesondere bei der Lösung des Gleichungssystems mittels Krylov-Unterraum Verfahren Anwendung.

Eine Möglichkeit der Vorkonditionierung ist die Wahl $\mathbf{P}_R = \mathbf{I}$ und $\mathbf{P}_R = \mathbf{A}^T$. Dies führt auf die sogenannte Normalgleichung

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}.$$

Die Matrix $\mathbf{A}^T\mathbf{A}$ ist immer symmetrisch und positiv definit, so dass zur Lösung des linearen Gleichungssystems das CG-Verfahren eingesetzt werden kann. Die Aufstellung der Normalgleichungen und ihre Kopplung mit dem CG-Verfahren wird in der Literatur auch als CGNR-Verfahren (CG Normal equations Residual minimizing) bezeichnet.

Ganz gezielt werden Vorkonditionierer zur Reduktion der Konditionszahl der Matrix eingesetzt. Ein idealer Vorkonditionierer wäre in diesem Zusammenhang z.B. die Wahl $\mathbf{P}_R = \mathbf{I}, \mathbf{P}_L = \mathbf{A}^{-1}$, da dann das umgeformte System die kleinste mögliche Konditionszahl 1 hätte und die Lösung des Gleichungssystems gefunden wäre. Weil $\mathbf{A}^{-1}$ in der Regel nicht leicht zu berechnen ist und oft einen hohen Speicherplatzbedarf benötigt, führt dies natürlich auf kein praktikables Verfahren. Dagegen kann man versuchen, eine Näherung für $\mathbf{A}^{-1}$ zu finden, da die umgeformte Koeffizientenmatrix dann nicht wesentlich von der Einheitsmatrix verschieden ist und somit eine geringe Konditionszahl hat. Für die explizit gegebenen Vorkonditionierer sollte diese Näherungsmatrix noch eine einfache Besetzungsstruktur haben, so dass die Vorkonditionierer einen geringen Speicherplatzbedarf aufweisen.

Der Vorkonditionierungsschritt muss während jeder Iteration eines iterativen Verfahrens in Form von Matrix-Vektor Multiplikationen durchgeführt werden. Die arithmetischen Operationen für die Vorkonditionierung dominieren in der Regel im gesamten iterativen Lösungsprozess.

In der Praxis stehen mehrere Vorkonditionierungsmethoden (siehe [2, 3]) zur Verfügung, wie z.B.

- Skalierung
- Polynomiale Vorkonditonierung
- Splitting-assoziierte Vorkonditionierung
- Unvollständige LU-Zerlegung
- Unvollständige Cholesky-Zerlegung
- Unvollständige QR-Zerlegung

Die verschiedenen Vorkonditionierungsverfahren haben unterschiedliche Einsatzbereiche. Im folgenden werden die Anwendungen der Vorkonditionierungstechniken auf das CG-Verfahren und auf das BiCGStab-Verfahren präsentiert. Die Splitting-asoziierte Vorkonditionierung und die unvollständige Cholesky-Zerlegung werden für das CG-Verfahren eingesetzt und für das BiCGStab-Verfahren verwendet man die unvollständige LU-Zerlegung.

3.5.1 Splitting-assoziierte Vorkonditionierung

Im Kapitel 2 wurden die Splitting-Methoden als Gleichungslöser vorgestellt. Gegenüber den Krylov-Unterraum Verfahren weisen sie ein schlechteres Konvergenzverhalten auf. Deshalb werden sie als iterative Lösungsverfahren kaum verwendet. Sie sind jedoch von großer Bedeutung, wenn man sie als Vorkonditionierer für die Krylov-Unterraum Verfahren einsetzt. Denn die Splitting-Methoden basieren auf einer Zerlegung $\mathbf{A} = \mathbf{M} - \mathbf{N}$, bei der $\mathbf{M}$ eine leicht invertierbare Approximation von $\mathbf{A}$ darstellt, ist es möglich, die Matrix $\mathbf{M}^{-1}$ als Vorkonditionierungsmatrix zu verwenden.

Definition: Ist durch die Iterationsvorschrift $\mathbf{x}_{m+1} := \mathbf{M}^{-1}\mathbf{N}\mathbf{x}_m + \mathbf{M}^{-1}\mathbf{b}$ eine Splitting-Methode zur Lösung von $\mathbf{A}\mathbf{x} = \mathbf{b}$ gegeben, so heißt $\mathbf{P} = \mathbf{M}^{-1}$ der zur Splitting-Methode assoziierte Vorkonditionierer.

Die Zerlegung von $\mathbf{A}$ ergibt eine Klasse von Splitting-assoziierten Vorkonditionierern, die in Tabelle 3.4 dargestellt werden. Dabei stellt $\mathbf{D} = diag\{a_{11}, \ldots, a_{nn}\}$ eine Diagonalmatrix dar. Ferner sind $\mathbf{L}$ und $\mathbf{R}$ die strikte linke untere bzw. die strikte rechte obere Dreiecksmatrix bezüglich $\mathbf{A}$.

Da in den Vorkonditionierern nur Diagonal- oder Dreiecksmatrizen vorkommen, kann die Vorkonditionierung somit implizit implementiert werden, d.h. die entsprechenden Eliminationstechniken werden bei den Matrix-Vektor Multiplikationen angewandt. Bei Produkten aus mehreren solchen Matrizen (wie beispielsweise $\mathbf{P}_{SGS}$) wird dies durch die Nacheinanderausführung dieser Techniken erreicht.

Die zum symmetrischen Gauss-Seidel und zum SSOR-Verfahren assoziierten Vorkonditionierungsmatrizen sind selbst wieder symmetrisch, wenn die Matrix $\mathbf{A}$ symmetrisch ist. Somit eignen sie sich insbesondere zur Vorkonditionierung des CG-Verfahrens.

$\mathbf{Splitting - Methode}$	$\mathbf{AssoziierteVorkonditionierungsmatrix}$
Jacobi Verfahren	$\mathbf{P}_J = \mathbf{D}^{-1}$
Gauss-Seidel Verfahren	$\mathbf{P}_{GS} = (\mathbf{L}+\mathbf{D})^{-1}$
Symm. Gauss-Seidel Verfahren	$\mathbf{P}_{SGS} = (\mathbf{D}+\mathbf{R})^{-1}\mathbf{D}(\mathbf{L}+\mathbf{D})^{-1}$
SOR-Verfahren	$\mathbf{P}_{SOR}(\omega) = \omega(\omega\mathbf{L}+\mathbf{D})^{-1}$
SSOR-Verfahren	$\mathbf{P}_{SSOR}(\omega) = \omega(2-\omega)(\mathbf{D}+\omega\mathbf{R})^{-1}\mathbf{D}(\omega\mathbf{L}+\mathbf{D})^{-1}$

Tabelle 3.4: Splitting-assoziierte Vorkonditionierer

3.5.2 Unvollständige Cholesky-Zerlegung

Die vollständige Cholesky-Zerlegung einer symmetrischen und positiv definiten Matrix $\mathbf{A}$ in der Form $\mathbf{A} = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$ ist als direktes Lösungsverfahren bekannt. Die unvollständige Formulierung dieser Zerlegung kann zur Vorkonditionierung ausgenutzt werden.

Die folgenden Begriffe (siehe [2], S. 197) sind hilfreich für die Herleitung der unvollständigen Cholesky-Zerlegung.

Definition: Die Menge

$$\mathcal{M} \subset \{(i,j)|i,j \in \{1,\ldots,n\}\}$$

heißt Matrixmuster in $\mathbb{R}^{n\times n}$. Für eine gegebene Matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ nennt man

$$\mathcal{M}^{\mathbf{A}} := \{(i,j)|a_{ij} \neq 0\}$$

das Besetzungsmuster von $\mathbf{A}$. Weiterhin heißt zu gegebenem Matrix- bzw. Besetzungsmuster $\mathcal{M}^{\mathbf{A}}$ die Menge

$$\mathcal{M}_S^{\mathbf{A}}(j) := \{i|(i,j) \in \mathcal{M}^{\mathbf{A}}\}$$

das zu $\mathcal{M}^{\mathbf{A}}$ gehörige j-te Spaltenmuster und

$$\mathcal{M}_Z^{\mathbf{A}}(j) := \{j|(i,j) \in \mathcal{M}^{\mathbf{A}}\}$$

das zu $\mathcal{M}^{\mathbf{A}}$ gehörige i-te Zeilenmuster.

Mit Hilfe des Besetzungsmusters lässt sich nun die unvollständige Cholesky-Zerlegung definieren (siehe [2], S. 200).

Definition: Sei $\mathbf{A} \in \mathbb{R}^{n\times n}$ symmetrisch und positiv definit, $\mathbf{L} = (l_{ij})_{i,j=1,\ldots,n} \in \mathbb{R}^{n\times n}$ eine reguläre linke untere Dreiecksmatrix. Die Zerlegung

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T + \mathbf{F} \tag{3.3}$$

existiere unter den Bedingungen

1. $l_{ij} = 0$, falls $(i,j) \notin \mathcal{M}^{\mathbf{A}}$,
2. $(\mathbf{L}\mathbf{L}^T)_{ij} = a_{ij}$, falls $(i,j) \in \mathcal{M}^{\mathbf{A}}$,

dann heißt (3.3) unvollständige Cholesky-Zerlegung (incomplete Cholesky, IC) der Matrix $\mathbf{A}$ zum Muster $\mathcal{M}^{\mathbf{A}}$. Die Matrix $\mathbf{F} \in \mathbb{R}^{n\times n}$ ist eine sogenannte Rest- oder Fehlermatrix. Vernachlässigt man die Matrix $\mathbf{F}$, so ergibt sich eine Matrix $\tilde{\mathbf{A}} = \mathbf{L}\mathbf{L}^T$, deren Inverse eine Approximation von $\mathbf{A}^{-1}$ darstellt und zur Vorkonditionierung des Gleichungssystems $\mathbf{A}\mathbf{x} = \mathbf{b}$ verwendet werden kann.

Aus der Definition ergibt sich die Berechnungsformeln der unvollständigen Cholesky-Zerlegung

$$l_{kk} = \sqrt{a_{kk} - \sum_{\substack{j=1 \\ j\in\mathcal{M}_Z^{\mathbf{A}}(k)}}^{k-1} l_{kj}^2}\,, \quad k = 1,\ldots,n$$

und

$$l_{ik} = \frac{1}{l_{kk}}\left(a_{ik} - \sum_{\substack{j=1 \\ j\in\mathcal{M}_Z^{\mathbf{A}}(i)\cap\mathcal{M}_Z^{\mathbf{A}}(k)}}^{k-1} l_{ij}l_{kj}\right), \quad i = k+1,\ldots,n \text{ und } i \in \mathcal{M}_Z^{\mathbf{A}}(k)$$

Da die Matrix $\mathbf{L}\mathbf{L}^T$ und das Produkt $\mathbf{L}\mathbf{A}\mathbf{L}^T$ symmetrisch und positiv definit sind, eignet sich die unvollständige Cholesky-Zerlegung sowohl für links- und rechts- als auch für beidseitige Vorkonditionierung, bei der es auf die Erhaltung dieser Eigenschaften ankommt, wie beispielsweise beim CG-Verfahren.

3.5.3 Das vorkonditionierte CG-Verfahren

Bei der Vorkonditionierung des CG-Verfahrens ist zu beachten, dass die Symmetrie und die positive Definitheit des Systems erhalten bleibt. Eine Möglichkeit dies zu gewähr-

leisten ist eine beidseitige Vorkonditionierung mit $\mathbf{P}_R = \mathbf{P}_L^T$. Somit erhält man eine transformierte Form des Gleichungssystems

$$\mathbf{My} = \mathbf{P}_L\mathbf{b}$$

mit $\mathbf{M} = \mathbf{P}_L\mathbf{A}\mathbf{P}_L^T$ und $\mathbf{y} = \mathbf{P}_L^{-T}\mathbf{x}$.

In Tabelle 3.5 ist der Algorithmus des vorkonditionierten CG-Verfahrens gegeben. Als Vorkonditionierer werden in den späteren Berechnungen $\mathbf{P}_{SGS}$ und $\mathbf{P}_{IC}$ eingesetzt. Da die Vorkonditionierung in der Iteration in Form einer Matrix-Vektor Multiplikation erfolgt und im $\mathbf{P}_{SGS}$ bzw. im $\mathbf{P}_{IC}$ nur Diagonal- und Dreiecksmatrizen auftreten, werden die entsprechenden Eliminationstechniken verwendet.

Bei PCG hat man, anders als beim normalen CG-Verfahren, keine Kontrolle des Residuums, denn die skalare Größe $\alpha_m = \|\mathbf{P}_L\mathbf{r}_m\|_2^2$ liefert keine Aussage darüber. Statt dessen wird in jedem Schritt der Wert $\|\mathbf{r}_m\|_2$ berechnet.

Wähle $\mathbf{x}_0 \in \mathbb{R}^n$
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{P}\mathbf{r}_0$, $\alpha_0 :=< \mathbf{r}_0, \mathbf{p}_0 >$
Für $m = 0, \ldots, n-1$

Y: $\|\mathbf{r}_m\|_2 \neq 0$	N
$\mathbf{v}_m := \mathbf{A}\mathbf{p}_m, \lambda_m := \dfrac{\alpha_m}{< \mathbf{v}_m, \mathbf{p}_m >}$	Exit
$\mathbf{x}_{m+1} := \mathbf{x}_m + \lambda_m\mathbf{p}_m$	
$\mathbf{r}_{m+1} := \mathbf{r}_m - \lambda_m\mathbf{v}_m$	
$\mathbf{z}_{m+1} := \mathbf{P}\mathbf{r}_{m+1}$	
$\alpha_{m+1} :=< \mathbf{r}_{m+1}, \mathbf{p}_{m+1} >$	
$\mathbf{p}_{m+1} := \mathbf{z}_{m+1} + \dfrac{\alpha_{m+1}}{\alpha_m}\mathbf{p}_m$	

Tabelle 3.5: Algorithmus des PCG-Verfahrens

3.5.4 Unvollständige LU-Zerlegung

Die unvollständige LU-Zerlegung, auch *incomplete LU-factorization (ILU)* genannt, spielt eine wichtige Rolle bei der Vorkonditionierung unsymmetrischer Gleichungssysteme. Die Idee dafür kommt aus der vollständigen LU-Zerlegung, die bei großen schwach-besetzten Matrizen durch *Fill-in* zu einer unerwünschten Erhöhung des Speicherplatzbedarfs führen könnte. Um dies zu vermeiden, stellt man eine Zerlegung der Matrix $\mathbf{A}$ in der Form

$$\mathbf{A} = \mathbf{LU} + \mathbf{F},$$

wobei $\mathbf{L}$ eine linke untere Dreiecksmatrix

$$\mathbf{L} = l_{ij} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix}$$

und $\mathbf{U}$ eine rechte obere Dreiecksmatrix

$$\mathbf{U} = u_{ij} = \begin{bmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & 1 & u_{23} & \cdots & u_{2n} \\ 0 & 0 & 1 & \cdots & u_{3n} \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

darstellt. $\mathbf{F}$ ist hierbei eine Rest- oder Fehlermatrix analog der unvollständigen Cholesky-Zerlegung. Die Matrizen $\mathbf{L}$ und $\mathbf{U}$ zusammen sollen soviel Speicherplatz wie die Matrix $\mathbf{A}$ belegen. Wird die Matrix $\mathbf{F}$ vernachlässigt, so ergibt sich eine leicht invertierbare Matrix $\tilde{\mathbf{A}} = \mathbf{LU}$, deren Inverse eine Approximation von $\mathbf{A}^{-1}$ darstellt und zur Vorkonditionierung des Gleichungssystems $\mathbf{Ax} = \mathbf{b}$ verwendet werden kann.

Definition: Sei $\mathbf{A} \in \mathbb{R}^{n\times n}$ eine reguläre Matrix, $\mathbf{L} = (l_{ij})_{i,j=1,\ldots,n} \in \mathbb{R}^{n\times n}$ eine reguläre linke untere Dreiecksmatrix und $\mathbf{U} = (u_{ij})_{i,j=1,\ldots,n} \in \mathbb{R}^{n\times n}$ eine reguläre rechte obere Dreiecksmatrix. Die Zerlegung

$$\mathbf{A} = \mathbf{LU} + \mathbf{F} \tag{3.4}$$

existiere unter den Bedingungen

1. $u_{ii} = 1$ für $i = 1, \ldots, n$,
2. $l_{ij} = u_{ij} = 0$, falls $(i, j) \notin \mathcal{M}^{\mathbf{A}}$,
3. $(\mathbf{LU})_{ij} = a_{ij}$, falls $(i, j) \in \mathcal{M}^{\mathbf{A}}$,

dann heißt (3.4) unvollständige LU-Zerlegung der Matrix $\mathbf{A}$ zum Muster $\mathcal{M}^{\mathbf{A}}$.

Zur Berechnung der i-ten Spalte $(l_{ii}, \ldots, l_{ni})^T$ von $\mathbf{L}$ stellen wir wegen $u_{mi} = 0\ (m > i)$ fest, dass

$$a_{ki} = \sum_{m=1}^{n} l_{km} u_{mi} = \sum_{m=1}^{i-1} l_{km} u_{mi} + l_{ki},$$

woraus

$$l_{ki} = \begin{cases} a_{ki} - \sum_{m=1}^{i-1} l_{km} u_{mi}, & \text{falls } (k,i) \in \mathcal{M}^{\mathbf{A}}, \\ 0, & \text{sonst,} \end{cases}$$

folgt. Anschließend erhalten wir die i-te Zeile $(u_{ii+1}, \ldots, u_{in})$ von $\mathbf{U}$ mit $l_{im} = 0\ (m > i)$ durch

$$a_{ik} = \sum_{m=1}^{n} l_{im} u_{mk} = \sum_{m=1}^{i-1} l_{im} u_{mk} + l_{ii} u_{ik},$$

mit

$$l_{ki} = \begin{cases} \frac{1}{l_{ii}}(a_{ik} - \sum_{m=1}^{i-1} l_{im} u_{mk}), & \text{falls } (i,k) \in \mathcal{M}^{\mathbf{A}}, \\ 0, & \text{sonst.} \end{cases}$$

Damit gewinnt man einen vorläufigen Algorithmus für die unvollständigen LU-Zerlegung, der in Tabelle 3.6 dargestellt ist. Dieser Algorithmus ist insofern vorläufig, als er noch nicht an das spezielle Speicherungsformat der Matrix $\mathbf{A}$ angepasst wird. Eine hieran angepasste Version des Algorithmus wird in Kapitel 4 vorgestellt.

Für $i = 1, \ldots, n$
Für $k = i, \ldots, n,\ \ k \in \mathcal{M}_S^{\mathbf{A}}(i)$
$l_{ki} = a_{ki} - \sum_{m=1}^{i-1} l_{km} u_{mi},\ \ m \in \mathcal{M}_Z^{\mathbf{A}}(k) \cap \mathcal{M}_S^{\mathbf{A}}(i)$
Für $k = i+1, \ldots, n,\ \ k \in \mathcal{M}_Z^{\mathbf{A}}(i)$
$u_{ik} = \frac{1}{l_{ii}}(a_{ik} - \sum_{m=1}^{i-1} l_{im} u_{mk}),\ \ m \in \mathcal{M}_Z^{\mathbf{A}}(i) \cap \mathcal{M}_S^{\mathbf{A}}(k)$

Tabelle 3.6: Vorläufiger Algorithmus der unvollständigen LU-Zerlegung

Definition (ILU-Vorkonditionierer): Ist die ILU-Zerlegung der Matrix $\mathbf{A}$ wie in Gl.(3.4) gegeben, so ist

$$\mathbf{P} := \mathbf{U}^{-1}\mathbf{L}^{-1}$$

die zugehörige Vorkonditionierungsmatrix.

Wie bei der Vorkonditionierung vom CG-Verfahren, wird die Vorkonditionierungsmatrix $\mathbf{P}$ nicht explizit berechnet. Stattdessen wird ihre Wirkung durch Vorwärts- und Rückwärtselimination erreicht.

Der Algorithmus der Vorkonditionierung ist in Tabelle 3.7 gegeben. Da die Matrix $\mathbf{P}$ die gleiche Besetzungsstruktur wie die Matrix $\mathbf{A}$ hat, werden die von Null verschiedenen Einträge von $\mathbf{P}$ im Vektor $\mathbf{f}$ der Länge $amount$ abgespeichert.

Für $i = 1, \ldots, n$
$x(i) = vec(i)$
Für $i = 1, \ldots, amount$
Y: $r(i) > c(i)$ — N
$x(r(i)) = x(r(i)) - f(i) * x(c(i))$
Y: $r(i) = c(i)$ — N
$x(r(i)) = x(r(i))/f(i)$
Für $i = amount, \ldots, 1$
Y: $r(i) < c(i)$ — N
$x(r(i)) = x(r(i)) - v(i) * x(c(i))$

Tabelle 3.7: Algorithmus der Vorkonditionierung mit unvollständiger LU-Zerlegung

3.5.5 Das vorkonditionierte BiCGStab-Verfahren

Die Konvergenz des vorkonditionierten CG-Verfahrens ist nur für symmetrische und positiv definite Matrizen gewährleistet. Zur Lösung von Systemen, die diese Bedingung nicht erfüllen, existiert jedoch eine Reihe von Modifikationen dieses Verfahrens.

In dieser Arbeit wird das vorkonditionierte BiCGStab-Verfahren (PBiCGStab) verwendet. Zur Vorkonditionierung wird die im vorherigen Abschnitt diskutierte unvollständige LU-Zerlegung eingesetzt. Der Algorithmus des vorkonditionierten BiCGStab-Verfahrens ist in der folgenden Tabelle gegeben, der eine spezielle Form dieses Verfahren darstellt. Die ausführliche Erläuterung dazu kann in [2] nachgelesen werden.

Wähle $\mathbf{x}_0 \in \mathbb{R}^n$ und $\varepsilon > 0$
$\mathbf{p}_0 := \mathbf{r}_0 := \mathbf{P}_L(\mathbf{b} - \mathbf{A}\mathbf{x}_0)$, $\rho_0 :=< \mathbf{r}_0, \mathbf{r}_0 >, j := 0$
Solange $\|\mathbf{r}_j\|_2 > \varepsilon$
$\mathbf{v}_j := \mathbf{A}\mathbf{P}_R\mathbf{p}_j, \tilde{\mathbf{v}}_j := \mathbf{P}_L\mathbf{v}_j$
$\alpha_j := \frac{\rho_j}{< \tilde{\mathbf{v}}_j, \mathbf{r}_0 >}, \mathbf{s}_j := \mathbf{r}_j - \alpha_j\mathbf{v}_j, \tilde{\mathbf{s}}_j := \mathbf{P}_L\mathbf{s}_j$
$\mathbf{t}_j := \mathbf{A}\mathbf{P}_R, \tilde{\mathbf{s}}_j, \tilde{\mathbf{t}}_j := \mathbf{P}_L\mathbf{t}_j$
$\omega_j := \frac{< \tilde{\mathbf{t}}_j, \tilde{\mathbf{s}}_j >}{< \tilde{\mathbf{t}}_j, \tilde{\mathbf{t}}_j >}$
$\mathbf{x}_{j+1} := \mathbf{x}_j + \alpha_j\mathbf{p}_j + \omega\tilde{\mathbf{s}}_j$
$\mathbf{r}_{j+1} := \mathbf{s}_j - \omega\mathbf{t}_j, \tilde{\mathbf{r}}_{j+1} := \tilde{\mathbf{s}}_j - \omega\tilde{\mathbf{t}}_j$
$\rho_{j+1} :=< \tilde{\mathbf{r}}_{j+1}, \mathbf{r}_0 >, \beta_j := \frac{\alpha_j}{\omega_j}\frac{\rho_{j+1}}{\rho_j}$
$\mathbf{p}_{j+1} := \tilde{\mathbf{r}}_{j+1} + \beta_j(\mathbf{p}_j - \omega\tilde{\mathbf{v}}_j), j = j + 1$
$\mathbf{x}_j := \mathbf{P}_R\mathbf{x}_j$

Tabelle 3.8: Algorithmus des vorkonditionierten BiCGStab-Verfahrens

Kapitel 4

Implementation moderner Gleichungslöser

In diesem Kapitel wird zuerst das Einbinden der iterativen Gleichungslöser in den vorliegende Finite-Elemente Code erläutert, wobei der angepasste Aufbau des linearen Gleichungssystems die wesentliche Aufgabe darstellt. Für symmetrische und unsymmetrische Koeffizientenmatrizen werden hierbei zwei unterschiedliche Algorithmen verwendet. Anschließend werden die implementierten Unterroutinen der Gleichungslöser vorgestellt.

4.1 Motivation

Da bei der FE-Diskretisierung oft sehr große lineare Gleichungssysteme entstehen, werden zur Einsparung von Speicherplatz nur die von Null verschiedenen Elemente der Koeffizientenmatrix (entspricht der tangentiellen Steifigkeitsmatrix eines Newton-ähnlichen Verfahrens) abgespeichert. Das verwendete Speicherformat wurde bereits im Abschnitt 2.4.1 vorgestellt, nämlich die Einträge werden zeilenorientiert in drei Vektoren abgespeichert. Der Vektor $\mathbf{v}$ registriert die Werte der Einträge und die anderen beiden Vektoren $\mathbf{r}$ und $\mathbf{c}$ beinhalten die Positionen der Einträge in der Koeffizientenmatrix. Alle drei Vektoren sind von der gleicher Dimension, gekennzeichnet mit $amount$, die die Anzahl der von Null verschiedenen Elemente der Koeffizientenmatrix angibt. Die zu implementierenden iterativen Gleichungslöser werden an dieses Speicherformat angepasst. Als Parameter werden die drei Vektoren beim Aufruf eines Gleichungslösers übergeben.

Der Aufbau dieser Vektoren erfolgt im FE-Code elementweise, d.h. sie setzen sich aus allen Elementsteifigkeiten zusammen. Ferner muss dieser Vorgang für jeden Zeit- und Lastschritt der FE-Berechnung wiederholt werden, da sich die tangentielle Steifigkeit ständig

verändert. Dies könnte bei feiner Vernetzung sehr aufwendig sein. Aus diesem Grund ist es sinnvoll, einen effizienten Algorithmus für die Assemblierung der tangentiellen Steifigkeit zu entwickeln. Im folgenden wird zunächst der implementierten Algorithmus für das PCG-Verfahren vorgestellt. Hierbei wird die Symmetrieeigenschaft der Koeffizientenmatrix ausgenutzt. Für das vorkonditionierte BiCGStab-Verfahren wird eine modifizierte Version dieses Algorithmus verwendet, wobei alle von Null verschiedenen Einträge der Koeffizientenmatrix abgespeichert werden müssen.

4.2 Aufbau des symmetrischen Gleichungssystems

Das PCG-Verfahren ist in der Regel nur für symmetrische und positiv definite Matrizen anwendbar. Aufgrund dieser Eigenschaft müssen nur die „Hälfe“ aller von Null verschiedenen Elemente abgespeichert werden, d.h. nur die linke untere Hälfte der Matrix wird berücksichtigt. Somit wird der Speicheraufwand weiter reduziert.

4.2.1 Bestimmung der Anzahl der zu speichernden Elemente

Da vor dem Aufruf eines Gleichungslösers der gesamte nötige Speicherplatz für dieses Verfahren vorzuhalten ist, muss die Dimension der drei Vektoren $\mathbf{v}$, $\mathbf{r}$ und $\mathbf{c}$, nämlich die Anzahl der zu speichernden Elemente, zuerst bestimmt werden. Außerdem ist an dieser Stelle ein Hilfsvektor der Dimension neq (Anzahl der Unbekannten des Gleichungssystems) einzuführen. In diesem Vektor, bezeichnet mit $\mathbf{saddr}$, werden die Positionen solcher Einträge im Vektor $\mathbf{v}$ registriert, die den ersten von Null verschiedenen Elementen jeder Zeile in der Koeffizientenmatrix entsprechen. Dies wird mit einem Beispiel verdeutlicht.

Beispiel: Gegeben sei die Koeffizientenmatrix $\mathbf{A}$

$$\mathbf{A} = \begin{bmatrix} \boxed{2} & & & & \\ \boxed{1} & 2 & & \text{sym.} & \\ 0 & \boxed{1} & 2 & & \\ 0 & 0 & \boxed{1} & 2 & \\ 0 & 0 & 0 & \boxed{1} & 1 \end{bmatrix} \quad \Longrightarrow \quad \mathbf{v}: \underbrace{\boxed{2}\boxed{1}\boxed{2}\boxed{1}\boxed{2}\boxed{1}\boxed{2}\boxed{1}\boxed{1}}_{amount = 9},$$

dann ist der Hilfsvektor $\mathbf{saddr}$ wie folgt gebildet:

$$\mathbf{saddr}: \quad \underbrace{\boxed{1}\boxed{2}\boxed{4}\boxed{6}\boxed{8}}_{neq = 5}$$

Wie wir später sehen werden, wird der Aufbau von **r**, **c** und weiteren Hilfsvektoren mit Hilfe des Vektors **saddr** beschleunigt. Anhand der vorgegebenen sogenannten Lokalisierungsmatrix[1] **lm**, wo die Elementfreiheitsgrade spaltenweise für alle Elemente den Strukturfreiheitsgraden zugeordnet sind, lässt sich der Hilfsvektor **saddr** mittels eines hier entwickelten Suchalgorithmus bestimmen. Hierbei wird die Anzahl der von Null verschiedenen Einträge für jede Zeile der Koeffizientenmatrix gezählt. Daraus ergeben sich die Werte von **saddr**. Die Bestimmung von $amount$ kann ohne zusätzlichen Aufwand in diesen Algorithmus integriert werden. Diese Berechnung an sich ist bei großen Gleichungssystemen schon aufwendig, da sie neq-mal über die Lokalisierungsmatrix laufen muss. Vorteil dabei ist jedoch, dass dieser Schritt während der gesamten FE-Berechnung nur einmal erfolgt, somit werden einige anderen wiederholt durchzuführenden Berechnungen viel effizienter.

Im folgenden wird der Programmabschnitt der Fortran-Implementation dargestellt. Hier bedeutet ldk Anzahl der Elementfreiheitsgrade und $nelem$ Anzahl der Elemente. Es wird im Algorithmus ein lokaler Hilfsvektor **h** der Dimension neq benötigt. Um Speicherplatz zu sparen, wird kein neuer Speicherplatz für **h** vorgehalten, sondern er benutzt den Speicherplatz eines anderen Vektors **allvec**, der später noch aufgebaut wird.

```
      amount = 0
      do i = 1, neq
!  ** set the comparison vector to zero
         do j = 1, i
           allvec(j) = 0
         enddo
!  ** compute the start address for the i-th row
         saddr(i) = amount + 1
!  ** search through the lm matrix for the i-th row
         do j = 1, nelem
           do k = 1, ldk
             if(lm(k,j).eq.i) then
               do l = 1, ldk
!  ** only the lower left part is considered
                 if(lm(l,j).le.i) then
                   do m = 1, i
!  ** in case of overlapping just jump over
                     if(lm(l,j).eq.allvec(m)) then
                       exit
!  ** in case of new entry amount is updated
                     elseif(allvec(m).eq.0) then
```

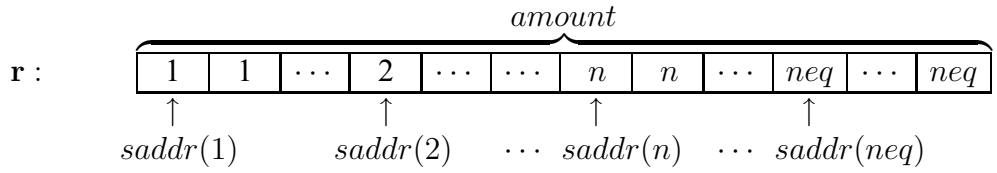
[1] Siehe [7], S. 92

```
                    allvec(m) = lm(l,j)
                    amount = amount + 1
                    exit
                  endif
                enddo
              endif
            enddo
            exit
          endif
        enddo
      enddo
    enddo
```

4.2.2 Aufbau des Indexfeldes

Die tangentielle Steifigkeitsmatrix, nämlich die Koeffizientenmatrix des Gleichungssystems, ändert sich ständig in jedem Iterationsschritt während der FE-Berechnung. Hierbei werden nur die Werte der Einträge verändert. Die Positionen, wo die von Null verschiedenen Einträge auftreten, sind stets gleichbleibend. Aufgrund dieser Eigenschaft muss das Indexfeld, nämlich **r** und **c**, nur einmal aufgebaut werden.

Da die Elemente zeilenorientiert abgespeichert werden, ist der Aufbau von **r** mit Hilfe von **saddr** besonders einfach. Die Stellen von $r(saddr(n))$ bis auf $r(saddr(n+1))$ werden alle mit n belegt, wobei $n = 1, \ldots, neq-1$ gilt. Für die letzte Zeile müssen die Stellen von $r(saddr(neq))$ bis zum Ende des Vektors mit neq belegt werden. Die Belegung sieht folgendermaßen aus:



Der Aufbau von **c** ist etwas komplizierter. Um Überlappung auszuschließen, muss der Spaltenindex eines neu gefundenen Beitrag aus der Elementsteifigkeit mit den bereits belegten Werten in **c** verglichen werden. Mit Hilfe von **saddr** wird jedoch der Vergleich auf einen bestimmten Bereich beschränkt, nämlich wo die Werte in **r** mit dem Zeilenindex dieses Beitrags übereinstimmen. Somit vermeidet man unnötige Rechenoperationen. Solange der Spaltenindex in diesem Bereich noch nicht vorhanden ist, wird er dann eingetragen. Es ist zu beachten, dass die Werte in diesem Bereich monoton steigend sein

müssen. Dies wird in Abb. 4.1 veranschaulicht.

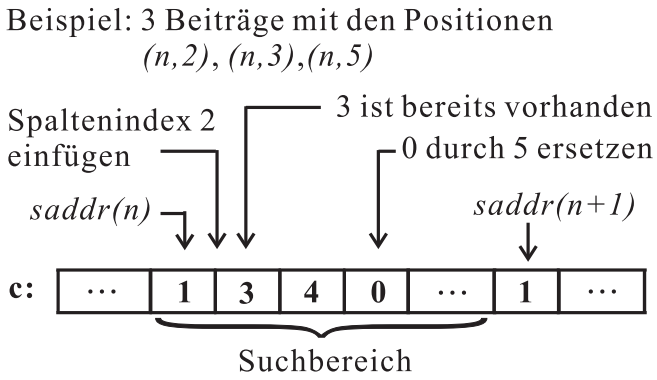


Abbildung 4.1: Aufbau des Spaltenindexfeldes

Im implementierten Fortran-Programm wird statt **r** und **c** ein Vektor **index** der Dimension $2 * amount$ definiert, in dem die erste Hälfte **r** entspricht und die zweite Hälfte für **c** benutzt wird.

4.2.3 Aufbau des Zuordnungsvektors

Die Werte der tangentiellen Steifigkeit ändert sich ständig. Das bedeutet, dass der Vektor **v** in jedem Zeit- und Lastschritt neu assembliert werden muss. Ohne Verwendung eines guten Algorithmus wird der Rechenaufwand bei großen Gleichungssystemen extrem groß. Dadurch könnte es dazu führen, dass der mögliche Zeitgewinn durch Einsatz eines geeigneten iterativen Gleichungslösers wieder aufgehoben wird.

Um den aufwändigen Suchvorgang zu vermeiden, wird für den Aufbau von **v** ein Zuordnungsvektor gebildet. Dieser Zuordnungsvektor mit der Dimension ne ($\geq amount$) stellt eine eindeutige Abbildung zu jedem nützlichen Beitrag aus den Elementsteifigkeiten und wird mit **allvec** bezeichnet. ne gibt die Anzahl aller nützlichen Beiträge aus den Elementsteifigkeiten an und ist wegen Positionsüberlappungen im Allgemeinen größer als $amount$. In diesem Vektor wird registriert, zu welchem Element in **v** ein Beitrag aus der Elementsteifigkeit gehört. Bei Positionsüberlappungen wird der Beitrag aus der Elementsteifigkeit zum entprechenden Element von **v** aufaddiert. Um die Eindeutigkeit der Abbildung zu gewährleisten, muss die gleiche Suchstrategie beim Aufbau von **allvec** sowie bei der Assemblierung von **v** verwendet werden. Ohne dies wird es zu fehlerhafter Assemblierung führen. Im folgenden wird der Programabschnitt zum Aufbau des Zuordnungsvektors gezeigt:

```
n = 1
do e = 1, nelem
  do i = 1, ldk
```

```
            irow = lm(i,e)
            if(irow.le.neq) then
              do m = 1, ldk
                icol = lm(m,e)
!   ** only the lower left part is considered
                if(icol.le.irow) then
                  do j = saddr(irow)+amount, 2*amount
!   ** if a new entry is found, its position is
!   ** written into allvec(n)
                    if(index(j).eq.icol) then
                      allvec(n) = j - amount
                      n = n + 1
                      exit
                    endif
                  enddo
                endif
              enddo
            endif
          enddo
        enddo
```

4.2.4 Assemblierung der tangentiellen Steifigkeit

Mit Hilfe des Zuordnungsvektors **allvec** ist die Arbeit zur Assemblierung der tangentiellen Steifigkeit, oder in der Fortran-Implementation die Bildung des Vektors **v**, besonders einfach, da auf einen mühsamen Suchalgorithmus verzichtet werden kann. Jeder gefundene Beitrag aus den Elementsteifigkeiten wird direkt einer bestimmtem Stelle im Vektor **v** zugeordnet. Wie zuvor bereits erwähnt wurde, muss der Algorithmus, wie man die Beiträge findet, jedoch der selbe sein wie bei der Bildung des Zuordnungsvektors. Der implementierten Programmabschnitt zur Assemblierung wird unten dargestellt. Diese Unterroutine gilt nur für eine Elementsteifigkeit, da sie im FE-Code für jede Elementsteifigkeit einmal aufgerufen wird.

```
!   ** the same algorithm to find a new entry
!   ** as for setting the vector allvec
        do i = 1, ldk
          irow = lm(i,e)
          if(irow.le.neq) then
```

```
            do j = 1, ldk
              icol = lm(j,e)
              if(icol.le.irow) then
!   ** new entry is added to the right position in v
!   ** k is a form parameter for the current address in
!   ** allvec and kelem is the element stiffness matrix
                v(allvec(k)) = v(allvec(k)) + kelem(i,j)
                k = k + 1
              endif
            enddo
          endif
!   ** set the right hand side for each row of
!   ** the system of linear equations
          rvec(irow) = rvec(irow) + relem(i)
        enddo
```

Der Aufbau des linearen Gleichungssystems erfolgt über die bisher diskutierten Schritte. Die wesentliche Aufgabe zur Assemblierung des Vektors **v** wurde relativ effizient durchgeführt, allerdings zu Last des Speicheraufwandes, da ein paar Hilfsvektoren eingesetzt wurden. Aufgrund der hohen Speicherkapazität heutiger Rechner ist in solchen Situationen meist die Ersparung der Rechenzeit heranzuziehen.

4.3 Aufbau des unsymmetrischen Gleichungssystems

Der Aufbau der unsymmetrischen Gleichungssysteme ist im Prinzip gleich wie bei den symmetrischen Gleichungssystemen. Der wesentliche Unterschied dabei liegt daran, dass alle von Null verschiedenen Einträge der tangentiellen Steifigkeitsmatrix abgespeichert werden müssen, weil keine Symmetriebedingung vorhanden ist.

Die im vorherigen Abschnitt erläuterten Algorithmen können weiter verwendet werden. Dazu sind jedoch bestimmte Modifikationen nötig. Die Einträge, bei denen $r(i) > c(i)$ ist, nämlich die rechte obere Hälfte der Matrix werden auch mit einbezogen. Der Programmabschnitt zur Assemblierung des Vektors **v** sieht dann folgendermaßen aus:

```
        do i = 1, ldk
          irow = lm(i,e)
          if(irow.le.neq) then
!   ** firstly assemble the diagonal elements
            v(allvec(k)) = v(allvec(k)) + kelem(i,i)
```

```
            k = k + 1
!   ** then assemble the non-diagonal elements
            do j = i+1, ldk
              icol = lm(j,e)
              if(icol.le.neq) then
                v(allvec(k)) = v(allvec(k)) + kelem(i,j)
                k = k + 1
!   ** exchange the indices of row and column for
!   ** the element for the symmetric position
                v(allvec(k),1) = v(allvec(k)) + kelem(j,i)
                k = k + 1
              endif
            enddo
          endif
!   ** generate the right hand side
          rvec(irow) = rvec(irow) + relem(i)
        enddo
```

Der Aufbau des Zuordnungsvektors **allvec** muss die gleiche Vorgehensweise einhalten. Wird hierbei ein Nebendiagonalbeitrag gefunden, so wird der andere Nebendiagonalbeitrag auf der symmetrischen Position sofort einsortiert.

4.4 Fortran-Implementationen der Gleichungslöser

Wie in Kapitel 3 diskutiert wurde, kann die Konvergenz des CG- und des BiCGStab-Verfahrens durch Vorkonditionierung beschleunigt werden. Es ist deshalb für praktische Aufgabenstellungen sinnvoll, nur die vorkonditionierten iterativen Gleichungslöser in den FE-Code zu implementieren. Für symmetrische Aufgabenstellungen wird das mit unvollständiger Cholesky-Zerlegung oder symmetrischem Gauss-Seidel Verfahren vorkonditionierte CG-Verfahren implementiert. Die Implementation für unsymmetrische Aufgabenstellungen erfolgt über das links- oder rechtsvorkonditionierte BiCGStab-Verfahren. Der Vorkonditionierer hierfür ist die unvollständige LU-Zerlegung.

4.4.1 Implementation der vorkonditionierten CG-Verfahren

Das CG-Verfahren ist nur für symmetrische und positiv definite Matrizen anwendbar. Durch die Vorkonditionierung mit unvollständiger Cholesky-Zerlegung und symmetri-

schem Gauss-Seidel Verfahren werden die beiden Eigenschaften beibehalten. Das vorkonditionierte CG-Verfahren (PCG) wird nach dem in Tabelle 3.5 dargestellten Algorithmus programmiert. Im Anhang werden die Programme $pcgic.f$ und $pcgsgs.f$ aufgelistet, wobei die Unterroutinen für die beiden Varianten der Vorkonditionierung aufgerufen werden. Sowohl bei unvollständiger Cholesky-Zerlegung als auch beim symmetrischen Gauss-Seidel Verfahren tritt die Vorkonditionierung in Form einer Matrix-Vektor Multiplikation auf. Da die Vorkonditionierer sich aus Dreiecks- oder Diagonalmatrizen zusammensetzen, ist die Matrix-Vektor Multiplikation nichts anderes als eine Vorwärts- und Rückwärtselimination. Die Implementation zur Vorkonditionierung, beispielsweise mit unvollständiger Cholesky-Zerlegung, ist unten aufgelistet. Bei symmetrischem Gauss-Seidel Verfahren sieht die Implementation ähnlich aus.

```
!  ** solve ff(t)x=vec using forward and backward elimination
!  ** initialization
      do i = 1, n
        x(i) = vec(i)
      enddo
!  ** forward elimination for fy=vec
      do i = 1, amount
        if(r(i).gt.c(i)) then
          x(r(i)) = x(r(i)) - f(i) * x(c(i))
        else
          x(r(i)) = x(r(i)) / f(i)
        end if
      enddo
!  ** backward elimination for f(t)x=y
      do i = amount, 1, -1
        if(r(i).eq.c(i)) then
          x(r(i)) = x(r(i)) / f(i)
        else
          x(c(i)) = x(c(i)) - f(i) * x(r(i))
        end if
      enddo
```

Hier dient der Vektor **f** zur Speicherung der von Null verschiedenen Cholesky-Faktoren. Die Speicherung erfolgt wie immer zeilenorientiert. Dieser Vektor wird vor dem Aufruf des PCG-Verfahrens zur Verfügung gestellt. Im folgenden ist die Fortran-Implementation dafür.

```
      do i = 1, amount
        f(i) = v(i)
```

```
!  ** diagonal entries
        if(r(i).eq.c(i)) then
          do j = saddr(r(i)), i-1
             f(i) = f(i) - f(j)**2
          enddo
          f(i) = dsqrt(f(i))
!  ** non-diagonal entries
        else
          do j = saddr(r(i)), i-1
            do k = saddr(c(i)), saddr(c(i)+1)-2
              if(c(j).eq.c(k))  f(i) = f(i) - f(j)*f(k)
            enddo
          enddo
          f(i) = f(i) / f(saddr(c(i)+1)-1)
        endif
      enddo
```

4.4.2 Implementation der vorkonditionierten BiCGStab-Verfahren

Das jeweils mit unvollständiger LU-Zerlegung (ILU) links- oder rechtsvorkonditionierte BiCGStab-Verfahren wird nach dem in Tabelle 3.8 dargestellen Algorithmus in $lbcgst.f$ und $rbcgst.f$ implementiert, welche im Anhang aufgelistet sind. Der Algorithmus zur Vorkonditionierung wurde bereits im Abschnitt 3.5.4 ausführlich vorgestellt.

Der vorläufige Algorithmus zur Bestimmung der ILU-Faktoren wurde im Abschnitt 3.5.4 präsentiert. In der Fortran-Inplementation wird dieser Algorithmus an das Speicherformat angepasst. Da die Besetzungsstruktur von ILU identisch wie die Koeffizientenmatrix ist, genügt in der Implementierung eine Schleife über den Vektor **v**. Dies bedeutet, dass für jeden von Null verschiedenen Eintrag der Koeffizientenmatrix der entsprechende ILU-Faktor berechnet wird. Die Vorgehensweise wird durch den folgenden Programmabschnitt verdeutlicht.

```
      f(1) = v(1)
!  ** loop over the vector v
      do i = 2, amount
        f(i) = v(i)
!  ** lower triangular matrix l
        if(r(i).ge.c(i)) then
          do j = saddr(r(i)),i-1
```

```
            do k = saddr(c(j)), saddr(c(j)+1)-2
              if(c(k).eq.c(i))  f(i) = f(i) - f(j)*f(k)
            enddo
          enddo
!   ** upper triangular matrix u,
!   ** whose diagonal entries are
!   ** all 1 and not saved
        else
          do j = saddr(r(i)),i-1
            if(r(j).gt.c(j)) then
              do k = saddr(c(j)), saddr(c(j)+1)-2
                if(c(k).eq.c(i))  f(i) = f(i) - f(j)*f(k)
              enddo
            elseif(r(j).eq.c(j)) then
              f(i) = f(i)/f(j)
              exit
            endif
          enddo
        endif
      enddo
```

Bis dahin sind der Aufbau des Gleichungssystems sowie die Implementation der modernen iterativen Verfahren abgeschlossen, wobei zusätzliche Arbeit für einige Hilfsvektoren neben den zu untersuchenden Gleichungslösern geleistet wurde.

Kapitel 5

Vergleichende Studien

Es wird in diesem Kapitel das Lösungsverhalten der im Kapitel 3 genannten iterativen Verfahren anhand eines nichtlinearen Problems studiert. Sie werden dem vorliegenden Direktlöser UMFPACK gegenübergestellt. Zum Schluss werden wir einige Möglichkeiten diskutieren, die auf eine Verbesserung der Iterationsverhalten der Verfahren führen könnten.

5.1 Voruntersuchung

Der in Abb. 5.1 gezeigte Block aus einem hyperelastischen Material (siehe [12], S.70, Modell 14) mit den Materialparametern

$$c_{10} = 0.1788\text{MPa}, \quad c_{01} = 0.1958\text{MPa}, \quad c_{30} = 0.00367\text{MPa}$$

wird verschiebungsgesteuert belastet. Die Reibungen zwischen den Kontaktflächen werden vernachlässigt. Das 8-knotige Q1P0-Volumenelement (Dreifeldformulierung, siehe [12], S. 130) kommt bei der FE-Berechnung zum Einsatz. Aus der Raumdiskretisierung ergibt sich ein nichtlineares Gleichungssystem [12]. Bei der Lösung des Gleichungssystems findet man Anwendung des Newton Verfahrens [13], das im FE-Code implementiert ist. Innerhalb des Newton Verfahrens müssen konsekutiv mehrere linearen Gleichungssysteme gelöst werden. Dazu werden die vorgestellten iterativen Verfahren eingesetzt.

Die Berechnungen werden auf einem SIEMENS CELSIUS-670 PC mit 2 GHz Taktfrequenz durchgeführt. Die Symmetrie des Problems wird ausgenutzt und es wird nur 1/8 des Blocks berechnet. Die Belastung wird innerhalb von 20 Lastschritten auf den Endwert gebracht, wobei die Lastinkremente jeder Laststufe konstant sind. Die zur letzten Laststufe

gehörende verformte Konfiguration, die einer Zusammendrückung von 30% enspricht, ist in Abb. 5.2 dargestellt. Die Verteilung der Normalspannungen in vertikaler Richtung kann ebenfalls diesem Bild entnommen werden.

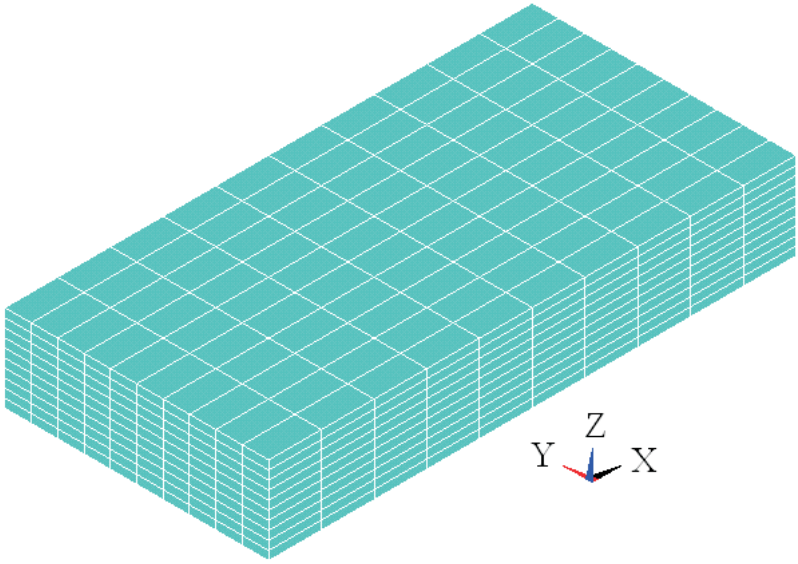


Abbildung 5.1: Dreidimensionales Beispiel für den Vergleich unterschiedlicher Lösungsverfahren

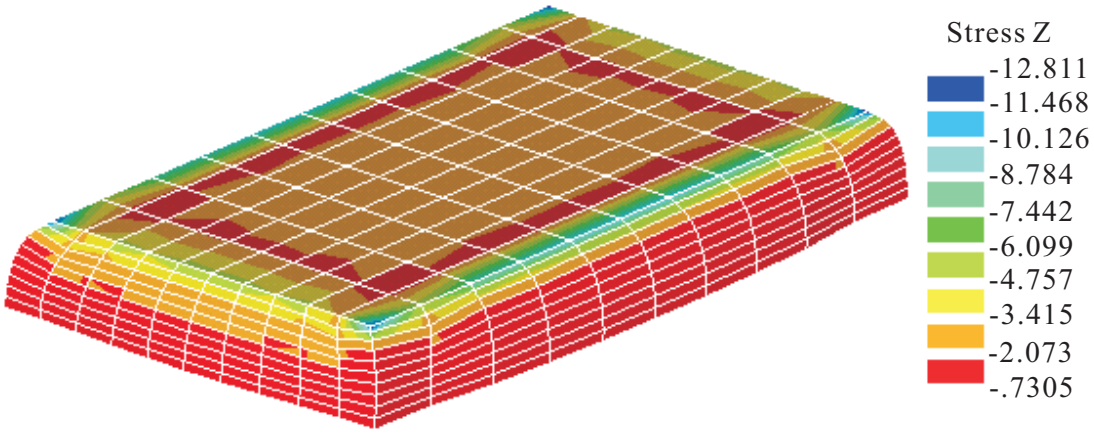


Abbildung 5.2: Deformierte Struktur und Normalspannungen in vertikaler Richtung

Aufgrund der Nahezu-Inkompressibilität des Materialmodells liegt der Wert des Kompressionsmoduls K (siehe [12], S. 76) üblicherweise zwischen 1000 – 2000MPa. Das Q1P0-Element liefert bei so hohem Wert von K erst nahe der Lösung des Newton Verfahrens lineare Gleichungssysteme mit positiv definiten Koeffizientenmatrizen, so dass das PCG-Verfahren nicht einsetzbar ist. Um die numersiche Untersuchungen fortsetzen zu können, wird das K im folgenden auf niedrigere Werte gesetzt.

5.2 Untersuchungen für das PCG-Verfahren

Da aus der in Abschnitt 5.1 vorgestellten Problemstellung nur symmetrische Gleichungssysteme entstehen, werden wir zunächst das PCG-Verfahren untersuchen. Als erstes wird ein lineares Gleichungssystem des Belastungsprozesses untersucht. Hierbei wird dasjenige zur halben äußeren Last herangezogen, welches im ersten Newton Iterationsschritt berechnet wird. Für diese Berechnung wird ein Kompressionsmodul $\mathrm{K} = 2\mathrm{MPa}$ gewählt. Dies entspricht mit

$$\nu = \frac{3\mathrm{K} - 2\mathrm{G}}{6\mathrm{K} + 2\mathrm{G}} \qquad \text{mit } \mathrm{G} = 1.573\mathrm{MPa}$$

einer Querkontraktionszahl $\nu = 0.18$ der bei der Ausgangskonfiguration linearisierten Materialgleichungen, wobei G den Schubmodul [15] darstellt. Diese Materialparameter weisen ein kompressibles Materialverhalten auf. Die Rechenzeit sowie die Anzahl der Iterationen der Verfahren für unterschiedliche Netzteilungen sind in Tabelle 5.1 zu finden.

Netzteilung	5x5x5	10x10x10	15x15x15	20x20x20
Anzahl der Unbekannten	444	3289	10784	25179
	Rechenzeit [s]			
PCG(SGS)	0.008	0.140	0.600	2.044
PCG(IC)	0.024	0.316	1.292	3.664
UMFPACK	0.024	1.952	26.630	–
	Iterationsanzahl			
PCG(SGS)	16	22	26	31
PCG(IC)	9	17	23	28

Tabelle 5.1: Rechenzeit und Iterationsanzahl für ein lineares Gleichungssystem

Hier sieht man deutlich, dass PCG(SGS) bei sehr kleinem K das beste Lösungsverhalten aufweist. Obwohl bei PCG(IC) Anzahl der Iterationen geringer ist, benötigt das Verfahren mehr Rechenzeit. Das liegt daran, dass vor der Iteration die unvollständige Cholesky-Zerlegung durchgeführt werden muss. Dies ist bei PCG(SGS) nicht der Fall, da zur Vorkonditionierung mit SGS lediglich die bereits vorhandenen Komponenten der Koeffizientenmatrix benötigt werden. Jedoch ist zu beachten, dass mit steigender Größe des Gleichungssystems das Verhältnis der Rechenzeiten von PCG(IC) und PCG(SGS) immer geringer wird. Es nimmt von 3 (bei 444 Unbekannten) auf 1.5 (bei 25179 Unbekannten) ab. Interessant ist noch zu finden, dass der Direktlöser UMFPACK fast bei allen Größen des Gleichungssystems viel mehr Rechenzeit aufweist. Allerdings ist es noch anzumerken, dass UMFPACK eigentlich für unsymmetrische Systeme konstruiert wird. Es zeigt sich, dass bei größeren Gleichungssystemen die Rechenzeit vom UMFPACK-Löser überproportional ansteigt. In Abb. 5.3 wird die Rechenzeit über Anzahl der Unbekannten veranschaulicht.

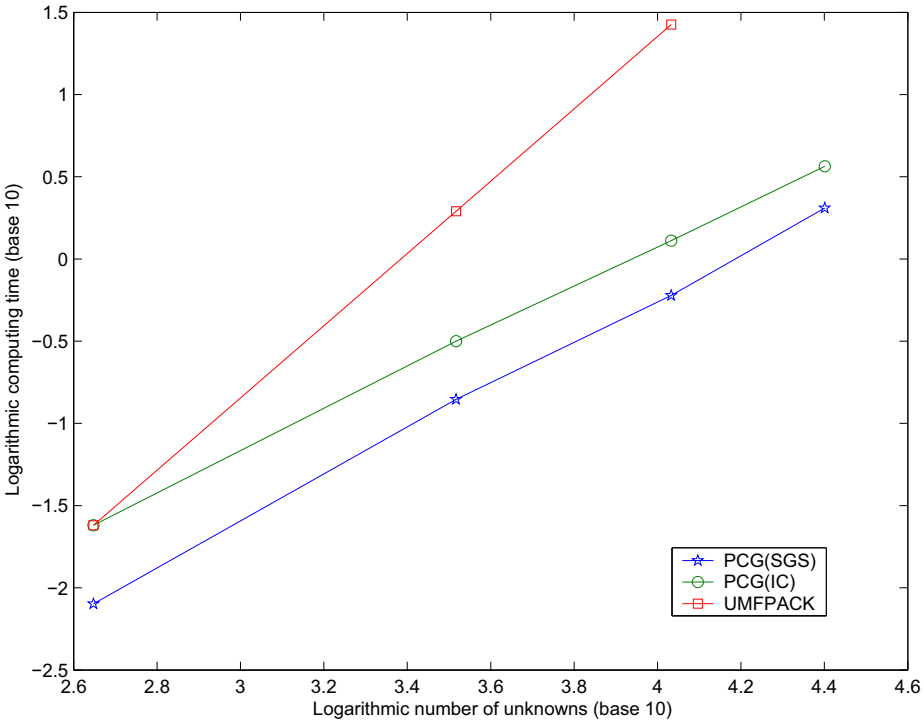


Abbildung 5.3: Verlauf der Rechenzeit über Anzahl der Unbekannten (PCG)

Mit steigendem K wird das Material immer inkompressibler und die Konditionszahl der zu lösenden Gleichungssysteme steigt. Als nächstes werden die Rechenzeit und Anzahl der Iterationen bei unterschiedlichen Werten von K für die Netzteilung 10x10x10 betrachtet, die in Tabelle 5.2 dargestellt sind. Das K wird bei dieser Untersuchung bis auf 20MPa gesteigert. Dies entspricht einer Querkontrakion $\nu = 0.46$. Bei K $= 30$MPa treten nicht positiv definite Matrizen auf, so dass die Berechnung des PCG-Verfahrens abgebrochen wird. Der Tabelle ist zu entnehmen, dass bei steigendem K alle drei Verfahren immer mehr Rechenzeit benötigen, was auf eine Verschlechterung der Kondition hindeutet. Außerdem zeigt die Anzahl der Iterationen bei PCG(SGS) und PCG(IC) mit steigendem K ebenfalls steigende Tendenz.

K [MPa]	1.05	2	5	10	15	20
	Rechenzeit [s]					
PCG(SGS)	0.120	0.140	0.144	0.156	0.176	0.180
PCG(IC)	0.320	0.316	0.336	0.340	0.360	0.384
UMFPACK	1.924	1.952	1.960	1.984	2.012	2.092
	Iterationsanzahl					
PCG(SGS)	20	22	24	25	27	29
PCG(IC)	18	17	19	21	24	31

Tabelle 5.2: Rechenzeit und Iterationsanzahl bei variierendem K ($n = 3289$)

Tabelle 5.3 registriert das Ergebnis einer gesamten FE-Berechnung für das gleiche Netz bei K $=$ 2MPa. Bei diesem Materialparameter und dieser Netzgröße benötigen alle drei Verfahren in jedem Lastschritt die gleiche Anzahl von Newton-Iterationen, d.h. es gibt bei jedem Verfahren die gleiche Anzahl von Gleichungssystemen zu lösen. In jedem Lastschritt benötigt PCG(SGS) insgesamt mehr Iterationen (in Tabelle 5.3 in Klammern angegeben) zur Lösung der Gleichungssysteme als PCG(IC). Wegen der aufwändigen unvollständigen Cholesky-Zerlegung für jedes Gleichungssystem benötigt PCG(IC) jedoch für die gesamte FE-Berechnung ungefähr doppelt soviel Zeit wie PCG(SGS), aber trotzdem ist PCG(IC) in diesem Fall effizienter gegenüber dem UMFPACK-Löser(6-fache Rechenzeit).

Lastschritt	Newton[PCG(SGS)]	Newton[PCG(IC)]	Newton[UMFPACK]
1	5[123]	5[94]	5
5	5[123]	5[93]	5
10	5[121]	5[93]	5
15	5[118]	5[95]	5
20	5[117]	5[94]	5
Zeit [s]	15.42	33.10	201.62

Tabelle 5.3: Gesamte Rechenzeit und Iterationsanzahl ($n = 3289$)

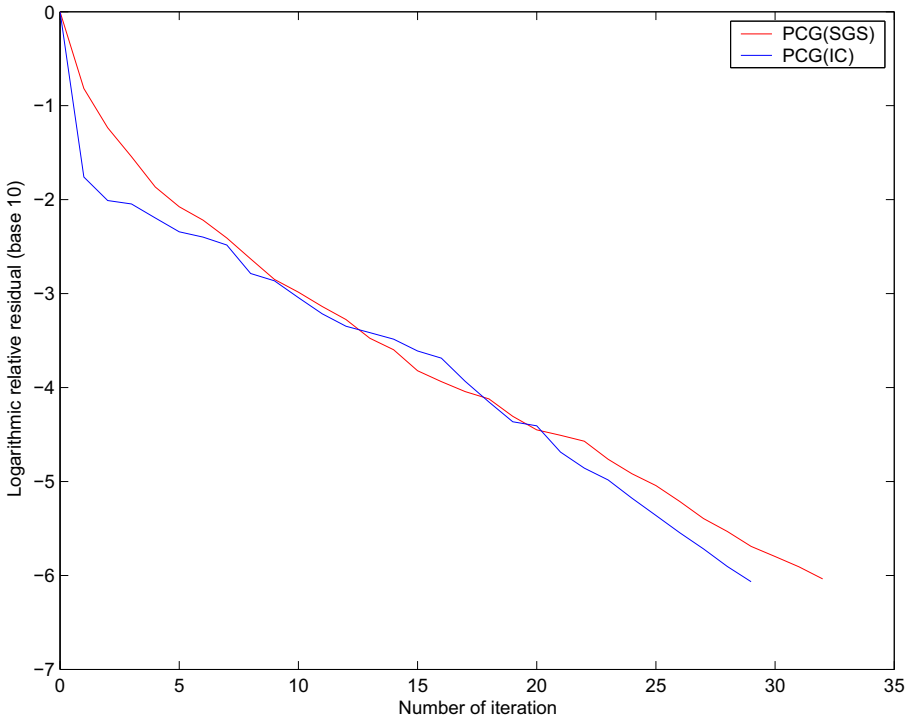


Abbildung 5.4: Verlauf des relativen Residuums (PCG, $n = 3289$)

Bei der Implementation des PCG-Verfahrens wird die Kontrolle des relativen Residuums $||\mathbf{r}_m||/||\mathbf{r}_0|| < 10^{-6}$ als Abbruchkriterium gewählt. Bei K $= 20$MPa wird der Verlauf des Residuums für das erste Gleichungssystem der FE-Berechnung in Abb. 5.4 dargstellt. Das Bild zeigt, dass beide Varianten des PCG-Verfahrens ähnliche Konvergenzverhalten aufweisen. Die Konvergenz ist quasi quadratisch, wobei PCG(IC) insgesamt geringere Iterationsschritte benötigt und PCG(SGS) ein glatteres Konvergenzverhalten aufweist.

5.3 Untersuchungen für das PBiCGStab-Verfahren

In diesem Abschnitt betrachten wir das für unsymmetrische Matrizen anwendbare vorkonditionierte BiCGStab-Verfahren. Erneut verwenden wir das gleiche Materialmodell. Jedoch werden die entstehenden Gleichungssysteme als „unsymmetrisch“ angenommen, da beim Aufbau der Gleichungssysteme für das PBiCGStab-Verfahren die vorhandene Symmetrieeigenschaft der Koeffizientenmatrix nicht berücksichtigt wird. Es werden hier die gleichen Untersuchungen wie beim PCG-Verfahren angestellt.

Zunächst betrachten wir das Lösungsverhalten des Gleichungssystems für die halbe Last mit K $= 2$MPa. In Tabelle 5.4 werden für unterschiedliche Netzteilungen die Rechenzeit und Anzahl der Iterationen eingetragen. Hier sieht man, dass bei kleinem Netz (444 Unbekannte) beide Varianten des vorkonditionierten BiCGStab-Verfahrens mehr Rechenzeit als UMFPACK-Löser benötigen. Mit steigender Größe des Gleichungssystems nimmt die Rechenzeit des UMFPACK-Lösers schnell zu und er benötigt dann mehr Zeit (siehe Abb. 5.5). Bei gleicher Größe des Gleichungssystems ist das PBiCGStab-Verfahren deutlich langsamer als das PCG-Verfahren. Dies liegt einerseits an dem Lösungsalgorithmus des PBiCGStab-Verfahrens, der komplizierter ist, andererseits an der für jedes Gleichungssystem durchzuführenden unvollständigen LU-Zerlegung, bei der mehr Arbeit geleistet werden muss.

Netzteilung	5x5x5	10x10x10	15x15x15	20x20x20
Anzahl der Unbekannten	444	3289	10784	25179
	Rechenzeit [s]			
BiCGStab(P_L)	0.108	1.616	7.188	60.188
BiCGStab(P_R)	0.104	1.524	6.784	35.566
UMFPACK	0.024	1.952	26.630	–
	Iterationsanzahl			
BiCGStab(P_L)	8	15	24	212
BiCGStab(P_R)	7	14	26	134

Tabelle 5.4: Rechenzeit und Iterationsanzahl für ein lineares Gleichungssystem

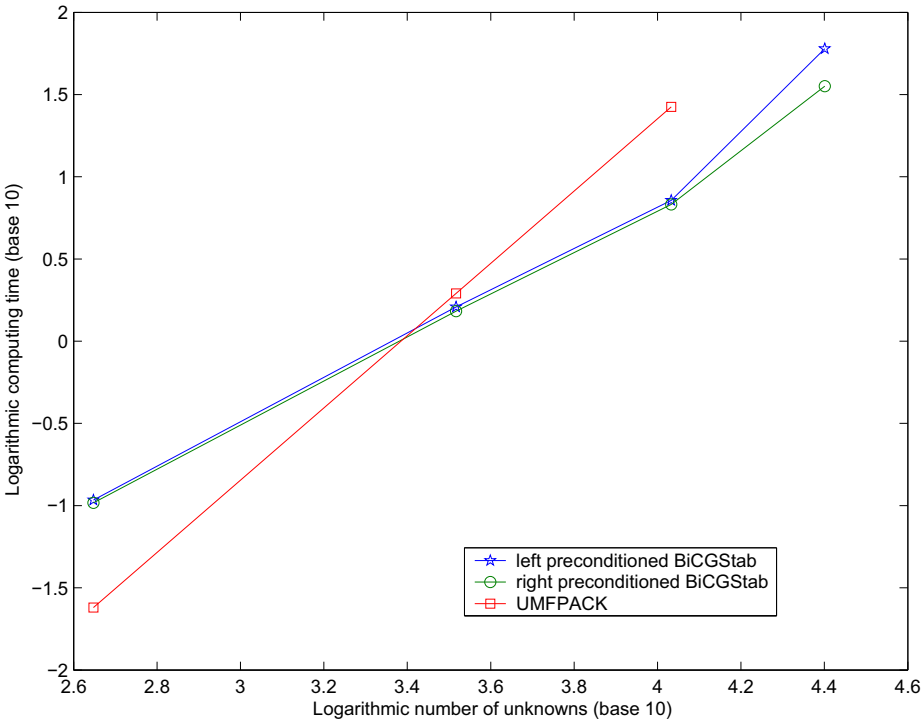


Abbildung 5.5: Verlauf der Rechenzeit über Anzahl der Unbekannten (PBiCGStab)

Als nächste Untersuchung werden die Rechenzeit und Anzahl der Iterationen bei unterschiedlichem K für die Netzteilung 10x10x10 in Tabelle 5.5 dargestellt.

K [MPa]	1.05	1.50	2	5
	Rechenzeit [s]			
BiCGStab(P_L)	1.536	1.548	1.616	–
BiCGStab(P_R)	1.484	1.488	1.524	1.724
UMFPACK	1.924	1.936	1.952	1.960
	Iterationsanzahl			
BiCGStab(P_L)	12	12	15	–
BiCGStab(P_R)	12	12	14	25

Tabelle 5.5: Rechenzeit und Iterationsanzahl bei variierendem K ($n = 3289$)

Mit größer werdendem K steigen die Rechenzeit und Anzahl der Iterationen wie bei den bereits diskutierten Verfahren an. Es ist anzumerken, dass das PBiCGStab-Verfahren im Vergleich mit dem PCG-Verfahren noch sensitiver gegenüber der Konditionszahl der Koeffizientenmatrix ist. Bei K $= 5$MPa (entspricht $\nu = 0.358$) konvergiert das linksvorkonditionierte BiCGStab-Verfahren schon nicht mehr. Bei K $= 10$MPa (entspricht $\nu = 0.425$) tritt beim rechtsvorkonditionierten BiCGStab-Verfahren Division durch Null auf, so dass die FE-Berechnung abgebrochen wird.

Das Ergebnis der gesamten FE-Berechnung für dieses Netz bei K $=$ 2MPa wird in Tabelle 5.6 dargestellt. Im Vergleich mit Tabelle 5.3 erkennt man, dass beide Varianten des PBiCGStab-Verfahrens bei großen Gleichungssystemen gegenüber dem UMFPACK-Löser eine effizientere Vorgehensweise aufweisen. Allerdings lassen sie sich nur für Matrizen mit niedriger Konditionszahl einsetzen. Unter gleichen Bedingungen liefert das rechtsvorkonditionierte BiCGStab-Verfahren bessere Lösungsverhalten als die linksvorkonditionierte Variante. Die Konvergenz des PBiCGStab-Verfahrens, wie in Abb. 5.6 gezeigt wird, ist nicht so glatt wie beim PCG-Verfahren.

Lastschritt	Newton[BiCGStab(P_L)]	Newton[BiCGStab(P_R)]
1	5[70]	5[71]
5	5[72]	5[71]
10	5[74]	5[73]
15	5[84]	5[83]
20	5[92]	5[90]
Zeit [s]	163.90	154.63

Tabelle 5.6: Gesamte Rechenzeit und Iterationsanzahl ($n = 3289$)

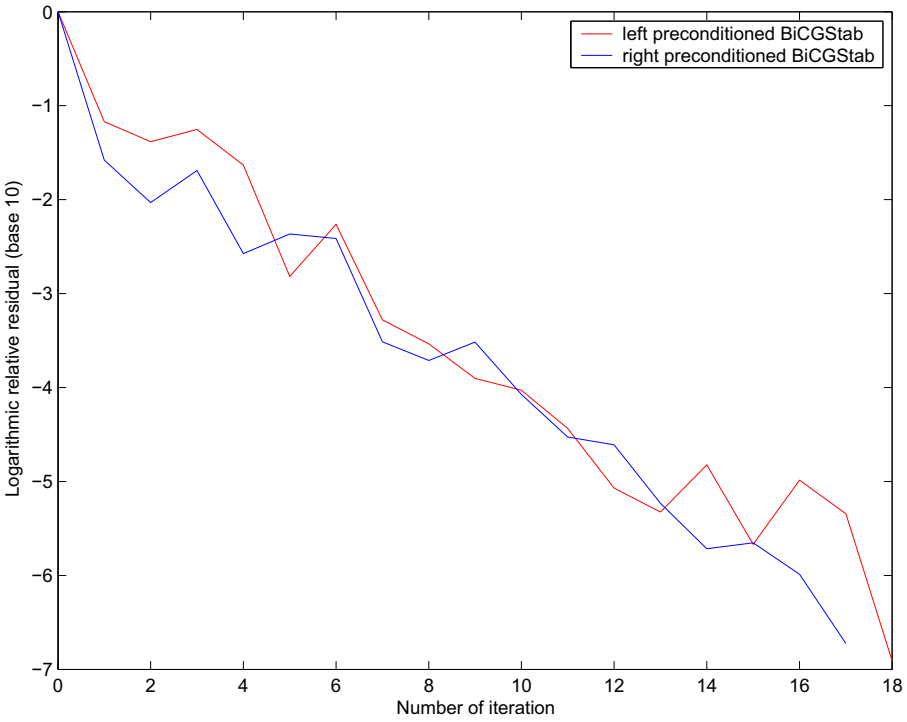


Abbildung 5.6: Verlauf des relativen Residuums (PBiCGStab, $n = 3289$)

5.4 Weitere Untersuchungen

Es besteht die Möglichkeiten, die implementierten iterativen Verfahren zu modifizieren, so dass bessere Lösungsverhalten erzielt werden könnten. Im folgenden werden wir zwei solcher Möglichkeiten untersuchen.

Die erste Modifikation bezieht sich auf die Wahl des Startvektors bei der iterativen Lösung des Gleichungssystems. Innerhalb eines Newton Verfahrens werden mehrere Gleichungssysteme gelöst. In bisherigen Implementationen wird der Startvektor für jedes Gleichungssystem auf den Nullvektor gesetzt. In der folgenden Untersuchung sind die Startvektoren so zu modifizieren, so dass für jedes neu zu lösende Gleichungssystem die Lösung des vorherigen Gleichungssystems als Startvektor angenommen wird. Der Startvektor für das erste Gleichungssystem des Newton Verfahrens muss dennoch beim Nullvektor bleiben.

PCG(SGS)		
Lastschritt	Newton-Iteration	
	Variante 1	Variante 2
1	5[121]	5[116]
5	5[119]	5[116]
10	5[116]	5[112]
20	5[120]	5[120]
Zeit [s]	15.77	15.47

PCG(IC)		
Lastschritt	Newton-Iteration	
	Variante 1	Variante 2
1	5[98]	5[95]
5	5[97]	5[93]
10	5[96]	5[93]
20	5[96]	5[97]
Zeit [s]	33.64	33.40

BiCGStab(P_L)		
Lastschritt	Newton-Iteration	
	Variante 1	Variante 2
1	5[84]	5[86]
5	5[92]	5[92]
10	5[106]	5[104]
20	5[270]	5[274]
Zeit [s]	187.09	187.34

BiCGStab(P_R)		
Lastschritt	Newton-Iteration	
	Variante 1	Variante 2
1	5[91]	–
5	5[92]	–
10	5[95]	–
20	5[259]	–
Zeit [s]	172.86	–

Tabelle 5.7: Lösungsvergleich bei Modifikation der Startvektoren

Tabelle 5.7 zeigt das Ergebnis der gesamten FE-Berechnung für das Netz 10x10x10 bei $K = 4$MPa, wobei Variante 2 die Lösung nach dieser Modifikation darstellt. Bei allen Verfahren bringt diese Modifikation leider keine Verringerung der Iterationsanzahl des Newton Verfahrens. Die Anzahl der Iterationen der Gleichungslöser in einem Lastschritt wird leicht verändert. Die Modifikation hat außerdem kaum Auswirkung auf die gesamte Rechenzeit der FE-Berechnung. Beim rechtsvorkonditionierten BiCGStab-Verfahren lieferte diese Vorgehensweise eine andere Lösung als PBiCGStab(P_L), was zu einem nicht konvergierenden Newton Verfahren führt. Eine Ursache konnte nicht herausgefunden werden.

Bisher wurde für die Konvergenz der iterativen Gleichungslöser das relative Residuum $||\mathbf{r}_m||/||\mathbf{r}_0|| < tol$ kontrolliert, wobei tol stets auf 10^{-6} gesetzt wurde. Man kann das Abbruchkriterium auch auf die Differenz der Lösungsvektoren zwischen zwei Iterationsschritten $||\mathbf{x}_{m+1} - \mathbf{x}_m||$ beziehen. Außerdem wird die Toleranz innerhalb eines Newton Verfahrens nicht mehr auf eine Konstante gesetzt, sondern sie wird mit der Toleranz der Newton-Iteration tol_n sowie der Norm des Lösungsvektors vom vorherigen Gleichungssystem $\|\mathbf{x}_{alt}\|$ assoziiert:

$$tol_x = min\{\alpha \cdot tol_n, \|\mathbf{x}_{alt}\|^2\} \qquad \text{mit} \quad \alpha = 0.1, tol_n = 10^{-3}$$

$$\Longrightarrow \qquad \|\mathbf{x}_{m+1} - \mathbf{x}_m\| \leq max\{tol_x, \beta \cdot tol_n\} \qquad \text{mit} \quad \beta = 0.001$$

Somit realisiert man ein veränderliches Abbruchkriterium. Am Anfang eines Newton Verfahrens sind die Rechenergebnisse von der gesuchten Nullstelle noch relativ weit entfernt und es werden dafür grobe Toleranzen eingesetzt. Im Lauf der Newton-Iterationen wird der Toleranzwert dann kleiner. Dies wird im FE-Code implementiert und Tabelle 5.8 zeigt uns die gesamten FE-Rechenzeiten der iterativen Verfahren bei K = 2MPa.

Rechenzeit [s]/PCG(SGS)		
Netzteilung	Toleranz	
	konstant	veränderlich
5x5x5	0.80	0.52
10x10x10	15.42	10.26

Rechenzeit [s]/PCG(IC)		
Netzteilung	Toleranz	
	konstant	veränderlich
5x5x5	2.23	2.18
10x10x10	33.10	30.19

Rechenzeit [s]/BiCGStab(P_L)		
Netzteilung	Toleranz	
	konstant	veränderlich
5x5x5	8.78	8.39
10x10x10	163.90	158.20

Rechenzeit [s]/BiCGStab(P_R)		
Netzteilung	Toleranz	
	konstant	veränderlich
5x5x5	8.40	8.40
10x10x10	154.63	151.74

Tabelle 5.8: Rechenzeiten bei veränderlicher Toleranz

Bei PCG(SGS) bringt das entwickelte Abbruchkriterium eine Ersparung der Rechenzeit um etwa 1/3. Dennoch ist es bei den anderen Verfahren nicht sehr wirkungsvoll, da die Rechenzeit nur geringfügig reduziert werden kann. Des Weiteren funktioniert es nicht bei jeder Netzteilung. Für beispielsweise das Netz mit 10784 Unbekannten werden in höheren Lastschritten die maximalen zulässigen Newton-Iterationen überschritten, so dass die Berechnung abgebrochen wird. Aus diesem Grund ist bei dieser Modifikation eine robuste Konstruktion des veränderlichen Abbruchkriteriums erforderlich.

Kapitel 6

Zusammenfassung und Ausblick

Diese Arbeit beschäftigt sich mit iterativen Verfahren zur Lösung von linearen Gleichungssystemen. In Kapitel 2 werden klassischen Verfahren, wie das Jacobi und das Gauss-Seidel Verfahren, vorgestellt und anhand eines Modellproblems ausgetestet. Die Konvergenzverhalten dieser Verfahren waren nicht zufriedenstellend, so dass sie nicht zur Lösung für FE-Anwendungen geeignet sind. Um solche großen linearen Gleichungssysteme schnell lösen zu können, benötigt man numerische Verfahren mit guten Konvergenzeigenschaften. Somit kamen Krylov-Unterraum Verfahren in Frage. Bei der Vielzahl von Krylov-Unterraum Verfahren werden das CG-Verfahren und das BiCGStab-Verfahren diskutiert, die für symmetrische bzw. unsymmetrische Systeme anwendbar sind. Da die Vorkonditionierung für iterative Gleichungslöser sehr wichtig ist, werden ebenfalls die vorkonditionierten Varianten der beiden Verfahren vorgestellt.

Des Weiteren braucht man wegen schwacher Besetztheit der Koeffizientenmatrix geeignete Datenstruktur, auf denen die Lösungsalgorithmen operieren. Dafür wird bei der Implementation eine spezielle Speichertechnik verwendet, nämlich Matrizen in Vektordarstellung, die nur von Null verschiedene Einträge enthält. Der Zusammenbau des linearen Gleichungssystems ist an diese Speichertechnik des FE-Codes angepasst worden.

Die Lösungsverhalten der implementierten Verfahren wird anhand eines hyperelastischen Materialmodells bei finiten Verzerrungen untersucht. Gegenüber dem UMFPACK-Löser stellen das PCG-Verfahren und das PBiCGStab-Verfahren effizientere Lösungsverfahren dar, wenn der Material ein stark kompressibles Verhalten aufweist. In diesem Fall hat das Gleichungssystem in der Regel geringe Konditionszahl. Je größer das Gleichungssystem ist, desto effizienter sind die iterativen Verfahren. Bei nahezu inkompressiblen Materialien ist die tangentielle Steifigkeitsmatrix (Koeffizientenmatrix) meist schlecht konditioniert und eventuell nicht mehr positiv definit. In diesem Fall treten bei den iterativen Gleichungslösern Schwierigkeiten auf. Im Abschluss der Arbeit werden zwei Varianten zur Modifikation der Fortran-Implementation untersucht. Bei der ersten Modifikation sind innerhalb des Newton Verfahrens die Startvektoren für die iterative Lösung der Glei-

chungssysteme (außer des ersten) nicht auf Nullvektoren gesetzt, sondern sie nehmen die Lösung des gerade gelösten Gleichungssystems an. Dies führt zu keiner Verbesserung der Lösungsverhalten. Bei der anderen Variante kam eine veränderliche Abbruchtoleranz des iterativen Lösers zum Einsatz, welche an die Genauigkeit des übergeordneten Newton Verfahrens angepasst ist. Dieses Abbruchkriterium müsste robust konstruiert werden, damit es für beliebige Netzteilungen funktionsfähig ist.

Aus diesen Betrachtungen ist sehr schwierig, eine allgemeingültige Aussage dazu zu treffen. Die iterativen Gleichungslöser können deshalb nur problemangepasst verwendet werden. Sie müssen an die Eigenschaften der Koeffizientenmatrix des linearen Gleichungssystems angepasst werden, bzw. bei realen nahezu inkompressiblen Materialmodellen bleibt die Anwendung eines geeigneten Verfahrens ein offenes Problem.

Anhang A

Quelltext der klassischen Verfahren

A.1 jacobi.f

Unterroutine zur Lösung des linearen Gleichungssystems mit Jacobi Verfahren:

```
      subroutine jacobi(n,amount,xnew,b,v,r,c,maxit,tol)
c
c*****************************************************
c     program-description:
c     --------------------
c     subroutine for solving system of linear
c     equations using jacobi method
c*****************************************************
c     7
      implicit none
      integer           n,i,j,amount,maxit,k
      integer           r(*),c(*)
      double precision  tol,sum,temp,conv
      double precision  b(*),x(n),xnew(*),v(*)
c
c  ** initialization
c
      k = 1
      conv = 1.d+0
      do i = 1,n
        x(i) = 0.d+0
        xnew(i) = 0.d+0
      enddo
c
      do
c
c  ** process of a new iteration
c
```

```
        if((k.le.maxit).and.(conv.gt.tol)) then
          j = 1
          do i = 1,n
            sum=b(i)
            do
              if(j.le.amount) then
                 if(r(j).eq.i) then
                    if(r(j).ne.c(j)) then
                       sum = sum - v(j) * x(c(j))
                    else
                       temp = v(j)
                    end if
                    j = j + 1
                 else
                    exit
                 end if
               else
                 exit
               end if
            enddo
            xnew(i) = sum / temp
          enddo
c
c  ** compute the relative error
c
          conv = 0.d+0
          do i = 1,n
            conv = conv + dble(abs(xnew(i) - x(i)))
            x(i) = xnew(i)
          enddo
          k = k + 1
        else
          exit
        end if
      enddo
c
      if(conv.le.tol) then
        print *, "jacobi iterative solution successful"
      else
        print *, "maximal iteration step reached"
      end if
      return
      end
```

A.2 gs.f

Unterroutine zur Lösung des linearen Gleichungssystems mit Gauss-Seidel Verfahren:

```
      subroutine gs(n,amount,x,b,v,r,c,maxit,tol)
c
```

```
c*****************************************************
c     program-description:
c     ____________________
c     subroutine for solving system of linear
c     equations using gauss-seidel method
c*****************************************************
c     7
      implicit none
      integer           n,i,j,amount,maxit,k
      integer           r(*),c(*)
      double precision  tol,sum,temp,conv
      double precision  b(*),x(*),xold(n),v(*)
c
c  ** initialization
c
      k = 1
      conv = 1.d+0
      do i = 1,n
        x(i) = 0.d+0
        xold(i) = 0.d+0
      enddo
c
      do
c
c  ** process of a new iteration
c
        if((k.le.maxit).and.(conv.gt.tol)) then
          j = 1
          do i = 1,n
            sum = b(i)
            do
              if(j.le.amount) then
                 if(r(j).eq.i) then
                    if(r(j).ne.c(j)) then
                       sum = sum - v(j) * x(c(j))
                    else
                       temp = v(j)
                    end if
                    j = j + 1
                 else
                    exit
                 end if
              else
                exit
              end if
            enddo
            x(i) = sum / temp
          enddo
c
c  ** compute the relative error
c
          conv = 0.d+0
          do i = 1,n
```

```
            conv = conv + dble(abs(x(i) - xold(i)))
            xold(i) = x(i)
          enddo
          k = k+1
        else
          exit
        end if
      enddo
c
      if(conv.le.tol) then
        print *, "gauss-seidel iterative solution successful"
      else
        print *, "maximal iteration step reached"
      end if
      return
      end
```

Anhang B

Quelltext der Krylov-Unterraum Verfahren

B.1 pcgsgs.f

Quellcode des PCG-Verfahrens mit symmetrischem Gauss-Seidel Verfahren als Vorkonditionierer:

```
      subroutine pcgsgs (n,amount,x,b,v,r,c,tol,maxit)
c
c*********************************************************************
c     program-description:
c     --------------------
c     subroutine for solving system of linear equations
c     using preconditioned cg method(pcg)
c     (preconditioner: sysmmetric gauss-seidel)
c*********************************************************************
c     7
      implicit none
      integer          i,m,n,amount,maxit
      double precision tol,tolr0
      double precision b(*),x(*),rr(n),p(n),vv(n),z(n)
      double precision v(*),alpha,lamda,temp,temp2
      integer          r(*),c(*)
c
c  ** initialization of x
c
      call cleard (x,n)
c
c  ** initialization of alpha,rr and p
c
      call mvmpcg(n,amount,x,v,r,c,vv)
```

```
      do i = 1, n
        rr(i) = b(i) - vv(i)
      enddo
      call presgs(n,amount,p,rr,v,r,c)
      call dotprd(n,rr,p,alpha)
c
      call dotprd(n,rr,rr,temp)
      tolr0 = tol * dsqrt(temp)
c
c  ** process of iteration
c
      do m = 1, maxit
         if(dsqrt(temp).ge.tolr0) then
           call mvmpcg(n,amount,p,v,r,c,vv)
           call dotprd(n,vv,p,temp)
           lamda = alpha / temp
           do i = 1, n
             x(i) = x(i) + lamda * p(i)
             rr(i) = rr(i) - lamda * vv(i)
           enddo
           call presgs(n,amount,z,rr,v,r,c)
           call dotprd(n,rr,z,temp)
           do i = 1, n
             temp2 = p(i) * temp / alpha
             p(i) = z(i) + temp2
           enddo
           alpha = temp
           call dotprd(n,rr,rr,temp)
         else
           exit
         end if
      enddo
c
      if(dsqrt(temp).gt.tolr0) then
        write (11,1005)
      else
        write (11,1010) m-1
      end if
 1005 format ('maximale Iterationsschritte erreicht')
 1010 format ('Anzahl der Iterationen m = ', i8)
      return
      end
```

B.2 pcgic.f

Quellcode des PCG-Verfahrens mit unvollständiger Cholesky-Zerlegung als Vorkonditionierer:

```
      subroutine pcgic (n,amount,x,b,v,r,c,f,tol,maxit)
c
```

```
c**********************************************************************
c     program-description:
c     ____________________
c     subroutine for solving system of linear equations
c     using preconditioned cg method(pcg)
c     preconditioner: incomplete cholesky factorization (ic)
c**********************************************************************
c     7
      implicit none
      integer           i,m,n,amount,maxit
      double precision  b(*),x(*),rr(n),p(n),vv(n),z(n),tol,tolr0
      double precision  v(*),f(*),alpha,lamda,temp,temp2
      integer           r(*),c(*)
c
c  ** initialization of x
c
      call cleard (x,n)
c
c  ** initialization of alpha,rr and p
c
      call mvmpcg(n,amount,x,v,r,c,vv)
      do i = 1, n
        rr(i) = b(i) - vv(i)
      enddo
      call preic(n,amount,p,rr,f,r,c)
      call dotprd(n,rr,p,alpha)
      call dotprd(n,rr,rr,temp)
      tolr0 = tol * dsqrt(temp)
c
c  ** process of iteration
c
      do m = 1, maxit
         if(dsqrt(temp).ge.tolr0) then
           call mvmpcg(n,amount,p,v,r,c,vv)
           call dotprd(n,vv,p,temp)
           lamda = alpha / temp
           do i = 1, n
             x(i) = x(i) + lamda * p(i)
             rr(i) = rr(i) - lamda * vv(i)
           enddo
           call preic(n,amount,z,rr,f,r,c)
           call dotprd(n,rr,z,temp)
           do i = 1, n
             temp2 = p(i) * temp / alpha
             p(i) = z(i) + temp2
           enddo
           alpha = temp
           call dotprd(n,rr,rr,temp)
         else
           exit
         end if
      enddo
c
```

```
      if(dsqrt(temp).gt.tolr0) then
        write (11,1005)
      else
        write (11,1010) m-1
      end if
 1005 format ('maximale Iterationsschritte erreicht')
 1010 format ('Anzahl der Iterationen m = ', i8)
      return
      end
```

B.3 lbcgst.f

Quellcode des linksvorkonditionierten BiCGStab-Verfahrens mit ILU als Vorkonditionierer:

```
      subroutine lbcgst(n,amount,x,b,v,r,c,f,tol,maxit)
c
c*********************************************************************
c     program-description:
c     --------------------
c     subroutine for solving system of linear equations
c     using bicgstab method with left-preconditioning (ilu)
c*********************************************************************
c     7
      implicit none
      integer          i,m,n,amount,maxit
      double precision tol,tolr0
      double precision b(*),x(*),rr(n),rrp(n),pp(n),vv(n),vvp(n)
      double precision r0p(n),normr,s(n),sp(n),t(n),tp(n),temp1,temp2
      double precision v(*),f(*),alphap,betap,omegap,rrop
      integer          r(*),c(*)
c
c  ** initialization of x
c
      call cleard (x,n)
c
c  ** initialization of rr,r0p,rrp,pp and rrop
c
      call mvmult(n,amount,x,v,r,c,rr)
      do i = 1, n
        rr(i) = b(i) - rr(i)
      enddo
      call dotprd(n,rr,rr,normr)
      tolr0 = tol * dsqrt(normr)
      call preilu(n,amount,r0p,rr,f,r,c)
      do i = 1, n
        rrp(i) = r0p(i)
        pp(i) = r0p(i)
      enddo
      call dotprd(n,rrp,rrp,rrop)
```

```
c
c  ** process of iteration
c
      do m = 1, maxit
         if(dsqrt(normr).gt.tolr0) then
           call mvmult(n,amount,pp,v,r,c,vv)
           call preilu(n,amount,vvp,vv,f,r,c)
           call dotprd(n,vvp,r0p,temp1)
           alphap = rrop / temp1
           do i = 1, n
             s(i) = rr(i) - alphap * vv(i)
           enddo
           call preilu(n,amount,sp,s,f,r,c)
           call mvmult(n,amount,sp,v,r,c,t)
           call preilu(n,amount,tp,t,f,r,c)
           call dotprd(n,tp,sp,temp1)
           call dotprd(n,tp,tp,temp2)
           omegap = temp1 / temp2
           do i = 1, n
             x(i) = x(i) + alphap * pp(i) + omegap * sp(i)
             rr(i) = s(i) - omegap * t(i)
             rrp(i) = sp(i) - omegap * tp(i)
           enddo
           call dotprd(n,rr,rr,normr)
           call dotprd(n,rrp,r0p,temp1)
           betap = alphap * temp1 / (omegap * rrop)
           rrop = temp1
           do i = 1, n
             pp(i) = rrp(i) + betap * (pp(i) - omegap * vvp(i))
           enddo
         else
           exit
         end if
      enddo
c
      if(dsqrt(normr).gt.tolr0) then
        write (11,1005)
      else
        write (11,1010) m-1
      end if
 1005 format ('maximale Iterationsschritte erreicht')
 1010 format ('Anzahl der Iterationen m = ', i8)
      return
      end
```

B.4 rbcgst.f

Quellcode des rechtsvorkonditionierten BiCGStab-Verfahrens mit ILU als Vorkonditionierer:

```
      subroutine rbcgst(n,amount,x,b,v,r,c,f,tol,maxit)
c
c*********************************************************************
c     program-description:
c     --------------------
c     subroutine for solving system of linear equations
c     using bicgstab method with right-preconditioning (ilu)
c*********************************************************************
c     7
      implicit none
      integer          i,m,n,amount,maxit
      double precision tol,tolr0
      double precision b(*),x(*),rr(n),p(n),vv(n),temp(n)
      double precision r0(n),normr,s(n),t(n),temp1,temp2
      double precision v(*),f(*),alpha,beta,omega,rro
      integer          r(*),c(*)

c
c  ** initialization of x
c
      call cleard(x,n)
c
c  ** initialization of rr,r0p,rrp,pp and rrop
c
      call mvmult(n,amount,x,v,r,c,r0)
      do i = 1, n
        r0(i) = b(i) - r0(i)
        rr(i) = r0(i)
        p(i) = r0(i)
      enddo
      call dotprd(n,r0,r0,normr)
      tolr0 = tol * dsqrt(normr)
      rro = normr
c
c  ** process of iteration
c
      do m = 1, maxit
         if(dsqrt(normr).gt.tolr0) then
           call preilu(n,amount,temp,p,f,r,c)
           call mvmult(n,amount,temp,v,r,c,vv)
           call dotprd(n,vv,r0,temp1)
           alpha = rro / temp1
           do i = 1, n
             s(i) = rr(i) - alpha * vv(i)
           enddo
           call preilu(n,amount,temp,s,f,r,c)
           call mvmult(n,amount,temp,v,r,c,t)
           call dotprd(n,t,s,temp1)
           call dotprd(n,t,t,temp2)
           omega = temp1 / temp2
           do i = 1, n
             x(i) = x(i) + alpha * p(i) + omega * s(i)
             rr(i) = s(i) - omega * t(i)
```

```
            enddo
            call dotprd(n,rr,rr,normr)
            call dotprd(n,rr,r0,temp1)
            beta = alpha * temp1 / (omega * rro)
            rro = temp1
            do i = 1, n
              p(i) = rr(i) + beta * (p(i) - omega * vv(i))
            enddo
          else
            exit
          end if
        enddo
c
c  ** compute the actual values of x
c
        call preilu(n,amount,x,x,f,r,c)
c
        if(dsqrt(normr).gt.tolr0) then
          write (11,1005)
        else
          write (11,1010) m-1
        end if
 1005 format ('maximale Iterationsschritte erreicht')
 1010 format ('Anzahl der Iterationen m = ', i8)
        return
        end
```

Literaturverzeichnis

[1] Wriggers, P. (2001). *Nichtlineare Finite-Elemente Methoden*. Springer Verlag, Berlin.

[2] Meister, A. (2005). *Numerik linearer Gleichungssysteme*. Vieweg Verlag, Wiesbaden, 2. Auflage.

[3] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. SIAM Society for Industial and Applied Mathematics, Philadelphia, 2nd edition.

[4] Axelsson, O. and Barker, V. A. (1984). *Finite Element Solution of Boundary Value Problems*. Academic Press, Inc., Orlando, Florida.

[5] Golub, G. and Van Loan, C. (1996). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition.

[6] Hackbusch, W. (1993). *Iterative Lösung großer schwachbesetzter Gleichungssysteme*. Teubner Verlag, Stuttgart.

[7] Hughes, Th.J.R. (1987). *The Finite Element Method*. Prentice Hall, Inc., Englewood Cliffs, New Jersey.

[8] Zienkiewicz, O.C. and Taylor, R.L. (1989). *The Finite Element Method*. McGraw Hill, New York.

[9] Bathe, K.-J. (2002). *Finite-Elemente-Methoden*. Springer Verlag, Berlin, 2., vollst. neu bearb. und erw. Aufl..

[10] Meißner, U. und Menzel, A. (2000). *Die Methode der finiten Elemente. Eine Einführung in die Grundlagen*. Springer Verlag, Berlin, 2. Auflage.

[11] van der Vorst, H.A. (1992). *BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems*. SIAM J. Sci. Stat. Comput., 13:631-644, 1992.

[12] Hartmann, S. (2003). *Finite-Elemente Berechnung inelastischer Kontinua: Interpretation als Algebro-Differentialgleichungssysteme*. Habilitationsschrift am Institut für Mechanik, Universität Kassel.

[13] Hartmann, S. (2005). *A remark on the application of the Newton-Raphson method in non-linear finite element analysis*. Comput. Mech., 36: 100-116, 2005, Springer Verlag, Berlin.

[14] Hartmann, S. (2006). *TASA-FEM: Ein Finite-Elemente Programm für raum-zeitadaptive gekoppelte Strukturberechnungen*. Version 1.0, Mitteilung 01/06 des Instituts für Mechanik, Universität Kassel.

[15] Haupt, P. (2000). *Einführung in die Mechanik: Technische Mechanik III*. Institut für Mechanik, Universität Kassel.