Lösungsverfahren und Vorkonditionierer in objekt-orientierter Implementierung für Konvektions-Diffusions-Reaktions-Probleme

Diplomarbeit

vorgelegt von
Nils Klimanis
aus
Goslar

angefertigt am
Institut für Numerische und Angewandte Mathematik
der Georg-August-Universität zu Göttingen
2001

Inhaltsverzeichnis

Einleitung

In den letzten Jahren hat ein Wissenschaftszweig der Numerischen Mathematik, das *Scientific Computing*, immer mehr Bedeutung erlangt. Hierbei untersucht man die rechnergestützte Lösung von Problemen, die beispielsweise aus den Naturwissenschaften und in zunehmendem Maße auch aus den Wirtschaftswissenschaften stammen und durch mathematische Modelle beschrieben werden.

Viele dieser Modelle werden mit partiellen Differentialgleichungen formuliert, wie zum Beispiel die *Konvektions-Diffusions-Reaktionsgleichungen* der Art

$$-\varepsilon\Delta u + \vec{b}\cdot\nabla u + cu = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{auf } \partial\Omega, \tag{2}$$

mit $\Omega \in \mathbf{R}^n$, die in der Strömungsphysik auftauchen. Da sich diese Gleichungen in der Regel bei in der Praxis auftauchenden Problemen der Lösbarkeit im klassischen Sinn entziehen (z.B. bei nichtglatten Randdaten), werden sie über eine Variationsformulierung in eine diskrete Problemstellung überführt, die zu einem linearen Gleichungssystem der Form

$$Ax = b \tag{3}$$

mit $x, b \in \mathbf{R}^n$ und dünnbesetzter Matrix $A \in \mathbf{R}^{n\times n}$ äquivalent ist. Je genauer man die kontinuierliche Lösung approximieren will, um so feiner hat man die Auflösung (Gitterweite) bei der Diskretisierung zu wählen, was zu entsprechend großen Dimensionen n der Koeffizientenmatrix führt. Da direkte Methoden zur Lösung von (3) bei diesen Größenordnungen aus Laufzeitgründen nicht in Frage kommen, ist man an schnellen iterativen Verfahren interessiert, die zumindest eine beliebig genaue Annährung an die diskrete Lösung erlauben.

Am Institut für Numerische und Angewandte Mathematik der Universität Göttingen wird nun mit **adr** (*„advection diffusion reaction“*) ein Programm zur numerischen Lösung von Gleichungen des Typs (1) entwickelt. Es erlaubt u.a. die Generierung eines Gitters auf dem Gebiet Ω, die Diskretisierung nach der Finite-Elemente-Methode und die Lösung des dabei entstehenden Gleichungssystems. Ziel dieser Diplomarbeit und mein Anteil an diesem Softwareprojekt war die Erstellung einer C++-Klassenbibliothek zur Lösung von linearen Gleichungssystemen. Ein wichtiger Aspekt war hierbei die Fragestellung, ob es mit den modernen Sprachmitteln einer objektorientierten Programmiersprache wie Templates, Iteratoren und Vererbung in C++ möglich ist, numerische Algorithmen effizient zu implementieren. Daß dies möglich ist und daß sich die Laufzeiten hierbei nicht vor denen von vergleichbaren Bibliotheken in C und Fortran zu verstecken brauchen, wurde beispielsweise bereits mit *Blitz++* (vgl. [Vel99]) angedeutet.

In Kapitel 1 dieser Arbeit werden die mathematischen Grundlagen von Konvektions-Diffusions-Reaktionsgleichungen sowie deren Variationsformulierung und Diskretisierung, wie sie in **adr** vorgenommen wird, zusammengetragen. Kapitel 2 beschäftigt sich mit linearen Gleichungssystemen und ihrer Lösung mit Hilfe von traditionellen Splitting-Verfahren und moderneren Krylovmethoden. Außerdem wird hier auf die Vorkonditionierung eingegangen.

In Kapitel 3 werden verschiedene Datenstrukturen vorgestellt, die zur Speicherung der Matrix A dienen. Die auf den Namen **lins** (*„library of iterative numerical solvers“*) getaufte Klassenbibliothek und ihre Implementierung wird in Kapitel 4 vorgestellt.

Im fünften Kapitel werden die Algorithmen und Vorkonditionierer anhand von Beispielen für verschiedene Diskretisierungsgrößen in bezug auf ihre Laufzeit verglichen. Das letzte Kapitel schließlich zieht ein Resümee und gibt einen Ausblick auf die weiteren Ansatzpunkte der Lösung von linearen Gleichungssystemen.

Kapitel 1

Mathematische Grundlagen

In diesem Kapitel werden die funktionalanalytischen Grundlagen vorgestellt, die nötig sind, um Diskretisierungsverfahren wie die Methode der finiten Elemente für lineare (elliptische) partielle Differentialgleichungen zu erklären. Da es sich hierbei um weitgehend bekannte bzw. elementare Tatsachen handelt, sind sie überwiegend ohne Beweis angegeben. Überall, wo dies der Fall ist, wird jedoch eine Quelle zitiert, aus der sich der entsprechende Nachweis entnehmen läßt.

1.1 Funktionenräume

1.1.1 Räume stetig differenzierbarer Funktionen

Wie allgemein üblich, wird mit Ω stets ein beschränktes Gebiet im $\mathbf{R}^n$ bezeichnet, mit $\overline{\Omega}$ dessen abgeschlossene Hülle und mit $\partial\Omega$ der Rand des Gebietes.
Die Mengen der auf Ω bzw. $\overline{\Omega}$ stetigen Funktionen heißen $C(\Omega)$ bzw. $C(\overline{\Omega})$.
Sei $\alpha := (\alpha_1, \ldots, \alpha_n)$ ein Multiindex der Länge $|\alpha| := \sum_{i=1}^{n} \alpha_i$. Dann werden die partiellen Ableitungen der Ordnung $|\alpha|$ einer entsprechend oft differenzierbaren Funktion $u : \Omega \to \mathbf{R}$ im Punkt $x \in \mathbf{R}$ folgendermaßen abgekürzt :

$$D^\alpha u(x) := \frac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \ldots \partial x_n^{\alpha_n}}(x), \quad |\alpha| \geq 1$$
$$D^{(0,\ldots,0)} u(x) := u(x).$$

Definition 1.1. *Ist $m \in \mathbf{N}_0$, dann sei $C^m(\Omega)$ die Menge der m-fach auf Ω stetig differenzierbaren Funktionen :*

$$C^m(\Omega) := \{v : \Omega \to \mathbf{R} | D^\alpha u \in C(\Omega), \forall \alpha : |\alpha| \leq m\}.$$

Entsprechend sei $C^m(\overline{\Omega})$ die Menge der Funktionen aus $C^m(\Omega)$ mit stetig auf $\overline{\Omega}$ fortsetzbaren Ableitungen bis zur Ordnung m.

Satz 1.2. *Sei $\overline{\Omega}$ kompakt. Dann ist $C^m(\overline{\Omega})$ ein Banachraum mit der Norm*

$$||u||_{C^m(\overline{\Omega})} := \max_{|\alpha| \leq m} \max_{x \in \overline{\Omega}} |D^\alpha u(x)|, \quad u \in C^m(\overline{\Omega}). \tag{1.1}$$

Beweis. Siehe Satz 5.4 aus [Lub98b] oder Lemma 1.8 in [Alt92]. □

Definition 1.3. *Die Menge der auf Ω beliebig oft stetig partiell differenzierbaren Funktionen wird mit $C^\infty(\Omega)$ bezeichnet.*

Definition 1.4 (Funktionen mit kompaktem Träger). *Mit $C_0^\infty(\Omega)$ bezeichnet man die Menge der unendlich oft differenzierbaren Funktionen mit kompaktem Träger in Ω. Sie wird folgendermaßen definiert :*

$$C_0^\infty(\Omega) := \{u \in C^\infty(\Omega) | \operatorname{supp} u \subset\subset \Omega\} \tag{1.2}$$

Definition 1.5. *Seien $0 < s \leq 1$ und $m \in \mathbf{N}_0$. Dann sei der Hölder-Raum $C^{m,s}(\overline{\Omega})$ die Menge der Funktionen $u \in C^m(\overline{\Omega})$, für die gilt*

$$||u||_{C^{m,s}(\overline{\Omega})} := ||u||_{C^m(\overline{\Omega})} + \sum_{|\alpha|=m} \sup_{\substack{x,y\in\overline{\Omega}\\ x\neq y}} \frac{|D^\alpha u(x) - D^\alpha u(y)|}{|x-y|^s} < \infty. \tag{1.3}$$

Satz 1.6. *Sei $\overline{\Omega}$ kompakt. Dann ist $C^{m,s}(\overline{\Omega})$ mit der in (1.3) definierten Norm ein Banachraum.*

Beweis. Siehe Lemma 1.8 in [Alt92]. □

1.1.2 L^p-Räume

Um die für die verallgemeinerten Lösungen partieller Differentialgleichungen benötigten Sobolev-Räume einzuführen, braucht man zunächst die Räume Lebesgue-integrierbarer Funktionen. Sei im folgenden $\Omega \subset \mathbf{R}^n$ eine (Lebesgue-)meßbare Punktmenge. Zwei Funktionen heißen äquivalent, wenn sie sich nur auf einer Menge vom Maß 0 unterscheiden.

Definition 1.7. *Sei $1 \leq p < \infty$. Dann wird die Menge aller Äquivalenzklassen meßbarer Funktionen $u : \Omega \to \mathbf{R}$ mit $L^p(\Omega)$ bezeichnet. Die zugehörige Norm ist definiert durch :*

$$||u||_{L^p(\Omega)} := \left(\int_\Omega |u(x)|^p \, dx\right)^{1/p} < \infty \tag{1.4}$$

Definition 1.8. *Die Menge aller Äquivalenzklassen der auf Ω wesentlich beschränkten Funktionen sei :*

$$L^\infty(\Omega) := \{u : \Omega \to \mathbf{R} \text{ meßbar } | \exists M < \infty : |u(x)| \leq M \text{ f.ü. in } \Omega\} \tag{1.5}$$

mit

$$||u||_{L^\infty(\Omega)} := \operatorname{ess\,max}_{x\in\Omega} |u(x)| = \operatorname{vrai\,max}_{x\in\Omega} |u(x)| := \inf M. \tag{1.6}$$

Satz 1.9. *Die Menge $L^p(\Omega)$ der Lebesgue-meßbaren Funktionen ist ein Banachraum mit der Norm*

$$||u||_{L^p(\Omega)} := \begin{cases} \left(\int_\Omega |u(x)|^p \, dx\right)^{\frac{1}{p}}, & 1 \leq p < \infty \\ \operatorname{vrai\,max}_{x\in\Omega} |u(x)|, & p = \infty \end{cases} \tag{1.7}$$

Beweis. Siehe Satz 6.15 in [Lub98b]. □

Bemerkung 1.10. *Für den Spezialfall $p = 2$ kann man auf $L^2(\Omega)$ ein Skalarprodukt durch*

$$(u, v)_\Omega = (u, v)_{L^2(\Omega)} := \int_\Omega u(x)v(x)dx, \quad u, v \in L^2(\Omega) \tag{1.8}$$

definieren. Hiermit wird der Raum $L^2(\Omega)$ zum Hilbert-Raum.

1.1.3 Verallgemeinerte Ableitungen und Sobolev-Räume

Bei vielen praktisch auftretenden Problemen reicht der klassische Lösungsbegriff nicht mehr aus, um zu befriedigenden Ergebnissen zu gelangen. Um die Lösbarkeitstheorie auch auf solche Probleme zu erweitern, führt man den Begriff der *schwachen* oder *verallgemeinerten Ableitung* und die *Sobolev-Räume* ein. Diese Hilfsmittel erlauben es, die Bedingungen an die Problemdaten zu lockern, wie man in Abschnitt 1.2.3 sehen wird.

Definition 1.11. *Die Menge der lokal Lebesgue-integrierbaren Funktionen wird mit*

$$L^1_{loc}(\Omega) := \{u : \Omega \to \mathbf{R} \text{ meßbar} \mid u \in L^1(A) \ \forall A \subset\subset \Omega\} \tag{1.9}$$

bezeichnet.

Definition 1.12 (Verallgemeinerte Ableitungen). *Eine Funktion $w_\alpha \in L^1_{loc}(\Omega)$ heißt* verallgemeinerte Ableitung *$D^\alpha u$ von $u \in L^1_{loc}(\Omega)$, wenn gilt*

$$\int_\Omega w_\alpha v \, dx = (-1)^{|\alpha|} \int_\Omega u D^\alpha v \, dx, \quad \forall v \in C_0^\infty(\Omega). \tag{1.10}$$

Definition 1.13 (Sobolev-Räume). *Sei $k \in \mathbf{N}$ und $1 \le p \le \infty$, dann heißt die Menge*

$$W^{k,p}(\Omega) := \{u \in L^p(\Omega) \mid D^\alpha u \in L^p(\Omega) \quad \forall \alpha : |\alpha| \le k\} \tag{1.11}$$

Sobolev-Raum *der Funktionen mit verallgemeinerten und zur p-ten Potenz integrierbaren Ableitungen bis zur Ordnung k.*

Satz 1.14. *Der Sobolev-Raum $W^{k,p}(\Omega)$ ist ein Banach-Raum bezüglich der Norm*

$$||u||_{W^{k,p}(\Omega)} := \begin{cases} \left(\sum_{|\alpha|\le k} ||D^\alpha u||^p_{L^p(\Omega)}\right)^{\frac{1}{p}}, & 1 \le p < \infty \\ \sum_{|\alpha|\le k} ||D^\alpha u||_{L^\infty(\Omega)}, & p = \infty \end{cases} \tag{1.12}$$

Beweis. Siehe z.B. Satz 7.25 in [Lub98b]. □

Für spätere Aussagen ist auch noch die Halbnorm

$$|u|_{W^{k,p}(\Omega)} := \Big(\sum_{|\alpha|=k} ||D^\alpha u||^p_{L^p(\Omega)}\Big)^{\frac{1}{p}} \tag{1.13}$$

von Bedeutung.

Definition 1.15. *Der Abschluß der Menge $C_0^\infty(\Omega)$ bezüglich der Norm $||\cdot||_{W^{k,p}(\Omega)}$ wird mit $W_0^{k,p}(\Omega)$ bezeichnet.*

In [GR94], Seite 83/84 wird gezeigt, daß es sich für Funktionen aus $W_0^{1,2}(\Omega)$ bei $|\cdot|_{W^{1,2}(\Omega)}$ und $||\cdot||_{W^{1,2}(\Omega)}$ um äquivalente Normen handelt.

Der Fall $p = 2$ wird für die spätere Anwendung der Sobolev-Räume auf die Lösbarkeitstheorie elliptischer partieller Differentialgleichungen noch eine wichtige Rolle spielen. Es gilt nämlich folgendes Resultat :

Satz 1.16. *Der Sobolev-Raum $W^{k,2}(\Omega)$ ist ein Hilbert-Raum mit dem Skalarprodukt*

$$(u, v)_{W^{k,2}(\Omega)} := \sum_{0 \le |\alpha| \le m} \int_\Omega D^\alpha u D^\alpha v \, dx. \tag{1.14}$$

Beweis. Siehe z.B. Satz 8.9 in [Lub98b]. □

1.2 Konvektions-Diffusions-Reaktions-Gleichungen

Die in dieser Arbeit untersuchte Klasse von Problemen sind Spezialfälle von elliptischen, linearen partiellen Differentialgleichungen, die erst einmal allgemein vorgestellt werden. Es wird auf die Lösbarkeit dieser Gleichungen eingegangen, und die dabei auftauchenden Schwierigkeiten werden geschildert.

1.2.1 Elliptische Randwertprobleme

Zunächst sei der allgemeine Fall einer linearen partiellen Differentialgleichung 2. Ordnung dargestellt.

Definition 1.17. *Die allgemeine Form einer linearen partiellen Differentialgleichung 2. Ordnung ist die Gleichung $Lu = f$, mit dem* Differentialoperator *L, der durch*

$$(Lu)(x) = -\sum_{i,j=1}^{n} a_{ij}(x) \frac{\partial^2 u(x)}{\partial x_i \partial x_j} + \sum_{i=1}^{n} b_i(x) \frac{\partial u(x)}{\partial x_i} + c(x)u(x) \tag{1.15}$$

definiert ist, wobei u eine Funktion $u = u(x) : \Omega \to \mathbf{R}$ ist. Die Funktionen

$$a_{ij}, b_i, c, f : \Omega \to \mathbf{R}, \quad i, j = 1, \ldots, n. \tag{1.16}$$

werden Problemdaten *genannt.*

Sei $a_{ij} \in C(\Omega)$ und $u(x) \in C^2(\Omega)$. Da dann die Reihenfolge der Differentiation keine Rolle spielt und da über sämtliche Funktionen a_{ij} aufsummiert wird, kann man ohne Einschränkungen voraussetzen, daß gilt $a_{ij} = a_{ji}$, was äquivalent dazu ist, daß die Matrix $A(x) := (a_{ij}(x))_{i,j=1}^n$ symmetrisch ist. Ist sie dies nicht, so führt die Matrix $\bar{A}(x) := \frac{1}{2}(A(x) + A^T(x))$ auf denselben Differentialoperator L.

Definition 1.18 (Elliptizität). *Für $x_0 \in \Omega$ seien $\lambda_i(x_0), i = 1, \ldots, n$ die Eigenwerte von $A(x_0)$. Dann heißt der Differentialoperator L* elliptisch *im Punkt x_0, falls $\lambda_i(x_0) \neq 0, i = 1, \ldots, n$ und alle Eigenwerte das gleiche Vorzeichen haben.*

Für eine eindeutige Lösbarkeit von (1.15) benötigt man zusätzliche Bedingungen an die Randwerte von $u(x)$, also etwa $(Bu)(x) = 0$ für $x \in \partial\Omega$ mit einem Operator B. Typische Probleme werden durch folgende Randbedingungen charakterisiert.

Definition 1.19 (Randbedingungen). *Sei $g(x), h(x) \in L^2(\partial\Omega)$, $\vec{\nu} = (\nu_i)_{i=1}^n \in \mathbf{R}^n$ der äußere Normaleneinheitsvektor an dem Gebietsrand $\partial\Omega$. Ist*

$$(Bu)(x) = u(x) - g(x), \tag{1.17}$$

so spricht man von einem Dirichletschen Randwertproblem, *bei $g(x) \equiv 0$ von einem* homogenen Dirichletschen Randwertproblem. *Bei den Randbedingungen*

$$(Bu)(x) = \frac{\partial u(x)}{\partial \vec{\nu}} - g(x) = \sum_{i=1}^{n} \frac{\partial u(x)}{\partial x_i}\nu_i - g(x) \tag{1.18}$$

heißt das Problem (1.15) Neumannsches Randwertproblem, *bei*

$$(Bu)(x) = \frac{\partial u(x)}{\partial \vec{\nu}} + h(x)u(x) - g(x) = \sum_{i=1}^{n} \frac{\partial u(x)}{\partial x_i}\nu_i + h(x)u(x) - g(x) \tag{1.19}$$

nennt man es Robinsches Randwertproblem.

1.2.2 Problemstellung

Die im folgenden untersuchten Spezialfälle von (1.15) sind die Konvektions-Diffusions-Reaktions-Gleichungen. Sie treten z.B. bei Stofftransport-Problemen in der Strömungsphysik auf.

Definition 1.20. *Sei $\Omega \subset \mathbf{R}^n$ ein offenes und beschränktes Gebiet. Als* Konvektions-Diffusions-Reaktions-Problem *bezeichnet man die folgende Fragestellung :*

Finde $u : \Omega \to \mathbf{R}$, so daß $Lu = f$ mit

$$Lu := -\varepsilon\Delta u + \vec{b} \cdot \nabla u + cu = f \quad \text{in } \Omega, \tag{1.20}$$

$$Bu = 0 \quad \text{auf } \partial\Omega, \tag{1.21}$$

und den folgenden Daten*:*

- *dem Diffusionsparameter $\varepsilon \in \mathbf{R}, \varepsilon > 0$,*
- *dem Strömungsfeld $\vec{b} = \vec{b}(x) = \big(b_i(x)\big)_{i=1}^n \in L^\infty(\Omega)^n$,*
- *dem Reaktionskoeffizienten $c = c(x) \in L^\infty(\Omega)$,*
- *der Funktion $f = f(x) \in L^2(\Omega)$*
- *und dem Operator B, der die Randdaten modelliert.*

Physikalisch interpretiert man z.B. hierbei die gesuchte Funktion u als Konzentration eines Stoffes in einem chemischen Reaktor, der durch das Gebiet Ω beschrieben wird. Der Ausdruck $-\varepsilon\Delta u$ beschreibt, wie sich u durch *Diffusion* verändert. Das ε bezeichnet man hierbei auch als Störungsparameter, der die Stärke der Diffusion bestimmt. Insbesondere für $\varepsilon \ll 1$ spricht man deshalb bei (1.20) oft auch von einer *singulär gestörten partiellen Differentialgleichung.* Der *Konvektionsterm* $\vec{b}\cdot\nabla u$ modelliert die Änderung von u durch die im Reaktor herrschende Strömung $\vec{b}$. Der Term $cu - f$ schließlich beschreibt die Änderung von u durch eine einfache chemische Reaktion.

1.2.3 Verallgemeinerte Lösungen und Lax-Milgram-Theorie

Die Überführung des Problems (1.20) in eine *Variationsgleichung* und die Anwendung der Lax-Milgram-Theorie hierauf erlauben es, die Anforderungen an die Problemdaten und insbesondere an die Lösung abzuschwächen.
Wesentlich ist hier zunächst der Begriff der *elliptischen Bilinearform.*

Definition 1.21. *Eine Bilinearform $a : X \times X \to \mathbf{R}$ bezeichnet man als X-elliptisch, falls es eine Konstante $\gamma > 0$ gibt, so daß*

$$a(v, v) \geq \gamma \, ||v||_X^2 , \quad \forall v \in X. \tag{1.22}$$

Lemma 1.22 (Lax-Milgram). *Sei X ein Hilbert-Raum. Des weiteren seien mit $a : X \times X \to \mathbf{R}$ eine stetige, X-elliptische Bilinearform und $f : X \to \mathbf{R}$ eine stetige Linearform erklärt.*
Dann existiert genau eine Lösung $u \in X$ der Variationsgleichung

$$Finde\ u \in X : \quad a(u, v) = f(v) \quad \forall v \in X. \tag{1.23}$$

Beweis. Vgl. Satz 13.6 in [Lub98b] oder Lemma 3.6 in [GR94]. □

Anhand des homogenen Dirichletschen Randwertproblems (1.20) wird nun die verallgemeinerte Aufgabenstellung eingeführt.
Multipliziert man die Gleichung (1.20) mit einer Funktion $v \in C_0^\infty(\Omega)$ und integriert anschließend über das gesamte Gebiet Ω, so erhält man

$$-\varepsilon \int_\Omega \sum_{i=1}^n \frac{\partial^2 u}{\partial x_i^2} v \, dx + \int_\Omega \sum_{i=1}^n b_i \frac{\partial u}{\partial x_i} v \, dx + \int_\Omega cuv \, dx = \int_\Omega fv \, dx. \tag{1.24}$$

Nun wendet man auf den ersten Term die Regel der partiellen Integration an :

$$\int_\Omega \sum_{i=1}^n \frac{\partial^2 u}{\partial x_i^2} v \, dx = \int_{\partial\Omega} \sum_{i=1}^n \frac{\partial u}{\partial x_i} v\nu_i \, dx - \int_\Omega \sum_{i=1}^n \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} \, dx \tag{1.25}$$

Das Integral über den Rand $\partial\Omega$ hat den Wert 0, da jede Funktion $v \in C_0^\infty(\Omega)$ auf dem Rand von Ω verschwindet. Weiterhin liegt $C_0^\infty(\Omega)$ dicht in $W_0^{1,2}(\Omega)$ (siehe Definition 1.15), weshalb man zu Elementen $u, v \in X := W_0^{1,2}(\Omega)$ übergehen kann.
Jetzt definiert man :

$$a(u, v) := \int_\Omega \Big(\varepsilon \sum_{i=1}^n \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i} + \sum_{i=1}^n b_i \frac{\partial u}{\partial x_i} v + cuv \Big) \, dx \tag{1.26}$$

$$f(v) := \int_\Omega fv \, dx \tag{1.27}$$

für die linke bzw. rechte Seite der Gleichung (1.24).

Definition 1.23. *Die aus den Termen (1.26) und (1.27) zusammengesetzte Variationsgleichung*

$$Finde\ u \in X: \quad a(u,v) = f(v) \quad \forall v \in X. \tag{1.28}$$

heißt verallgemeinerte Aufgabenstellung *oder* schwache Formulierung *des Problems (1.20) mit homogenen Dirichletschen Randbedingungen.* $u \in X$ *heißt* verallgemeinerte *oder* schwache Lösung *von (1.20). Die Funktionen* v *nennt man* Testfunktionen.

Um das Lemma von Lax-Milgram auf die Gleichung (1.28) anwenden und damit Aussagen über ihre Lösbarkeit machen zu können, muß nachgewiesen werden, daß es sich bei $a(\cdot,\cdot)$ um eine stetige, X-elliptische Bilinearform und bei $f(\cdot)$ um eine stetige Linearform handelt. Hiermit befassen sich die beiden folgenden Lemmata.

Lemma 1.24. *Seien* $a(\cdot,\cdot)$ *und* $f(\cdot)$ *wie in (1.26) und (1.27) definiert, mit* $X := W_0^{1,2}(\Omega)$. *Für* $\varepsilon, \vec{b}, c$ *und* f *gelten die in Definition (1.20) gemachten Voraussetzungen. Dann ist* $a(\cdot,\cdot)$ *auf* $X \times X$ *eine stetige (beschränkte) Bilinearform und* $f(\cdot)$ *auf* X *eine stetige (beschränkte) Linearform.*

Beweis. Es sei im folgenden stets $u, v \in W_0^{1,2}(\Omega)$.

(i.) Die Linearität von $f(\cdot)$ und die Bilinearität von $a(\cdot,\cdot)$ prüft man durch einfaches Nachrechnen. Sie folgen aus der Linearität des Lebesgue-Integrals.

(ii.) Die Beschränktheit von $f(\cdot)$ folgt aus der Hölderschen und der Friedrichschen Ungleichung sowie aus der Tatsache, daß es sich bei $|\cdot|_{W^{1,2}(\Omega)}$ und $||\cdot||_{W^{1,2}(\Omega)}$ um äquivalente Normen handelt :

$$\begin{aligned} |f(v)| \underset{H\ddot{o}lder}{\leq} ||f||_{L^2(\Omega)}\, ||v||_{L^2(\Omega)} &\underset{Friedrich}{\leq} C_F\, ||f||_{L^2(\Omega)}\, |v|_{W^{1,2}(\Omega)} \\ &\leq C\, ||f||_{L^2(\Omega)}\, ||v||_{W^{1,2}(\Omega)} \end{aligned}$$

(iii.) Zum Nachweis der Beschränktheit von $a(\cdot,\cdot)$ betrachtet man die drei Terme einzeln.

(a.) Zunächst schätzt man den Diffusionsterm folgendermaßen ab :

$$\begin{aligned} \left| \varepsilon \int_\Omega \sum_{i=1}^n \frac{\partial u}{\partial x_i} \frac{\partial v}{\partial x_i}\, dx \right| &\underset{H\ddot{o}lder}{\leq} \varepsilon \sum_{i=1}^n \left|\left| \frac{\partial u}{\partial x_i} \right|\right|_{L^2(\Omega)} \left|\left| \frac{\partial v}{\partial x_i} \right|\right|_{L^2(\Omega)} \\ &\underset{\substack{Cauchy- \\ Schwarz}}{\leq} \varepsilon \sqrt{\sum_{i=1}^n \left|\left| \frac{\partial u}{\partial x_i} \right|\right|^2_{L^2(\Omega)}} \sqrt{\sum_{i=1}^n \left|\left| \frac{\partial v}{\partial x_i} \right|\right|^2_{L^2(\Omega)}} \\ &= \varepsilon\, |u|_{W^{1,2}(\Omega)}\, |v|_{W^{1,2}(\Omega)} \leq \varepsilon\, ||u||_{W^{1,2}(\Omega)}\, ||v||_{W^{1,2}(\Omega)}\,. \end{aligned}$$

(b.) Setzt man $\vec{b}_{max} = \left(b_{max,i}\right)_{i=1}^{n} \in \mathbf{R}^n$ mit $b_{max,i} := ||b_i||_{L^\infty(\Omega)}$, so ergibt sich für den Konvektionsterm die Abschätzung

$$\begin{aligned}
\left|\int_\Omega \sum_{i=1}^{n} b_i \frac{\partial u}{\partial x_i} v \, dx\right| &\underset{H\ddot{o}lder}{\leq} \sum_{i=1}^{n} ||b_i||_{L^\infty(\Omega)} \left|\left|\frac{\partial u}{\partial x_i}\right|\right|_{L^2(\Omega)} ||v||_{L^2(\Omega)} \\
&\underset{\substack{Cauchy-\\ Schwarz}}{\leq} \sqrt{\sum_{i=1}^{n} ||b_i||^2_{L^\infty(\Omega)}} \sqrt{\sum_{i=1}^{n} \left|\left|\frac{\partial u}{\partial x_i}\right|\right|^2_{L^2(\Omega)}} ||v||_{L^2(\Omega)} \\
&\leq \left|\left|\vec{b}_{max}\right|\right|_2 |u|_{W^{1,2}(\Omega)} ||v||_{L^2(\Omega)} \\
&\leq \left|\left|\vec{b}_{max}\right|\right|_2 ||u||_{W^{1,2}(\Omega)} ||v||_{W^{1,2}(\Omega)} .
\end{aligned}$$

(c.) Schließlich gilt noch :

$$\left|\int_\Omega cuv \, dx\right| \underset{H\ddot{o}lder}{\leq} ||c||_{L^\infty(\Omega)} ||u||_{L^2(\Omega)} ||v||_{L^2(\Omega)} \leq ||c||_{L^\infty(\Omega)} ||u||_{W^{1,2}(\Omega)} ||v||_{W^{1,2}(\Omega)} .$$

Zusammengefaßt ergibt dies also :

$$|a(u,v)| \leq C \, ||u||_{W^{1,2}(\Omega)} ||v||_{W^{1,2}(\Omega)} \quad \forall u, v \in W^{1,2}(\Omega)$$

mit $C = \varepsilon + \left|\left|\vec{b}_{max}\right|\right|_2 + ||c||_{L^\infty(\Omega)}$.

□

Lemma 1.25. *Erfüllen die Problemdaten aus Definition (1.20) überdies noch die Voraussetzungen*

$$\frac{\partial b_i}{\partial x_i} \in L^\infty(\Omega), \; i = 1, \ldots, n \quad \textit{also} \quad \vec{b} \in W^{1,\infty}(\Omega)^n$$

und

$$c - \frac{1}{2} \sum_{i=1}^{n} \frac{\partial b_i}{\partial x_i} = c - \frac{1}{2} \nabla \cdot \vec{b} \geq c_0 > 0,$$

so ist die Bilinearform $a(\cdot,\cdot)$ X-elliptisch.

Beweis. Wendet man auf den Konvektionsterm die Regel der partiellen Integration an, so erhält man, da auch hier wieder das Randintegral wegfällt, die Beziehung

$$\int_\Omega \sum_{i=1}^{n} b_i \frac{\partial v}{\partial x_i} v \, dx = -\frac{1}{2} \int_\Omega \frac{\partial b_i}{\partial x_i} v^2 \, dx. \tag{1.29}$$

Eingesetzt in $a(v,v)$ gilt damit

$$\begin{aligned}
a(v,v) &= \varepsilon \sum_{i=1}^{n} \left|\left|\frac{\partial v}{\partial x_i}\right|\right|^2_{L^2(\Omega)} + \int_\Omega \underbrace{\left(c - \nabla \cdot \vec{b}\right)}_{\substack{\geq c_0 \\ \text{n. V.}}} v^2 \, dx \\
&\geq \varepsilon \, |v|^2_{W^{1,2}(\Omega)} + c_0 \, ||v||^2_{L^2(\Omega)} \geq C \, ||v||^2_{W^{1,2}(\Omega)}
\end{aligned}$$

mit $C = \min(\varepsilon, c_0)$. □

Damit ist Folgendes gezeigt :

Satz 1.26. *Unter den in Definition (1.20), Lemma (1.24) und Lemma (1.25) gemachten Voraussetzungen existiert genau eine verallgemeinerte Lösung des Randwertproblems (1.20).*

Beweis. Ergibt sich durch Anwendung des Lemmas von Lax-Milgram. □

Bemerkung 1.27. *Wie man im Beweis zu Lemma (1.25) sehen kann, stützt sich die Elliptizität der Bilinearform $a(\cdot,\cdot)$ und damit die Lösbarkeit der Variationsgleichung (1.28) im Sinne der Lax-Milgram-Theorie auf die Tatsache, daß ε und $c_0 > 0$ sind. Man kann die in Lemma (1.25) gemachte Voraussetzung jedoch insoweit abschwächen, als man lediglich fordert, daß $c - \frac{1}{2}\nabla \cdot \vec{b} \geq 0$ gilt. Man kann dann im Beweis mit der Normäquivalenz von $|\cdot|_{W^{1,2}(\Omega)}$ und $||\cdot||_{W^{1,2}(\Omega)}$ argumentieren. Dies ist für konkrete Probleme aus der Praxis auch notwendig, denn häufig betrachtet man inkompressible Strömungen, für die $\nabla \cdot \vec{b} \equiv 0$ gilt, und endotherme chemische Reaktionen, bei denen $c \geq 0$ ist. In diesen Fällen (insbesondere bei $c \equiv 0$) basiert die Elliptizität von $a(\cdot,\cdot)$ allerdings auch nur darauf, daß $\varepsilon > 0$ ist.*

1.3 Diskretisierungsverfahren

Um die Variationsformulierung (1.28) numerischen Lösungsverfahren zugänglich zu machen, ist man nun daran interessiert, die Fragestellung in ein diskretes Problem zu überführen. Hierbei versucht man, Näherungslösungen in endlichdimensionalen Teilräumen zu finden. Nachfolgend werden zwei Verfahren vorgestellt, zum einen die Standard-Finite-Elemente-Methode (FEM) vom Ritz-Galerkin-Typ, die sich für jegliche elliptischen Probleme eignet, und zum anderen eine stabilisierte FEM, die speziell an die bei Konvektions-Diffusions-Reaktions-Problemen auftretenden Schwierigkeiten angepasst ist.

1.3.1 Ritz-Galerkin-Verfahren

Die Idee der in diesem Abschnitt beschriebenen Verfahren ist es, die Lösung $u \in X$ durch ein Element $u^n \in X_n$ eines endlich dimensionalen Unterraumes

$$X_n \subset X, \quad \dim X_n = n < \infty$$

zu approximieren.

Definition 1.28. *Seien die Bilinearform $a(\cdot,\cdot)$ und die Linearform $f(\cdot)$ wie in Definition (1.23) gegeben. Dann nennt man das Problem*

$$\text{Finde } u^n \in X_n : \quad a(u^n, v) = f(v) \quad \forall v \in X_n. \tag{1.30}$$

Ritz-Galerkin-Verfahren *zur Variationsgleichung (1.28).*

Im nächsten Schritt läßt sich zeigen, daß das Problem (1.30) äquivalent zu einem linearen Gleichungssystem ist. Ist $\{\phi_i\}_{i=1}^n$ eine Basis von X_n, dann läßt sich jedes $v \in X_n$

als Linearkombination

$$v = \sum_{i=1}^{n} v_i \phi_i, \quad \text{mit} \quad \vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \in \mathbf{R}^n \tag{1.31}$$

darstellen. Deshalb, und weil $a(\cdot,\cdot)$ und $f(\cdot)$ linear bezüglich ihrer Argumente sind, ist das Problem (1.30) dazu äquivalent, die Gleichheit $a(u^n, \phi_i) = f(\phi_i)$ lediglich für jedes Basiselement zu fordern.
Definiert man nun noch

$$A = (a_{ij})_{i,j=1}^{n} \in \mathbf{R}^{n\times n} \quad mit\ a_{ij} := a(\phi_j, \phi_i) \tag{1.32}$$

sowie

$$\vec{u} = (u_1, \ldots, u_n)^T \in \mathbf{R}^n, \quad mit\ u^n = \sum_{i=1}^{n} u_i \phi_i$$
$$\vec{f} = (f_1, \ldots, f_n)^T \in \mathbf{R}^n, \quad mit\ f_i := f(\phi_i),$$

dann gilt folgendes für $i = 1, \ldots, n$:

$$a(u^n, \phi_i) = \sum_{j=1}^{n} u_j a(\phi_j, \phi_i) = \sum_{j=1}^{n} a_{ij} u_j = f(\phi_i) = f_i.$$

Damit ist die nachfolgende Aussage bewiesen :

Lemma 1.29. *Das Ritz-Galerkin-Verfahren (1.30) ist äquivalent zu dem linearen Gleichungssystem*

$$A\vec{u} = \vec{f}. \tag{1.33}$$

Die Matrix A wird *Steifigkeitsmatrix* genannt. Weiterhin ist nun von Interesse, ob und wann sich ein solches Gleichungssystem lösen läßt. Benutzt man das Standardskalarprodukt im $\mathbf{R}^n$, $\langle \vec{u}, \vec{v} \rangle := \sum_{i=1}^{n} u_i v_i$, dann kann man schreiben :

$$a(u, v) = \langle A\vec{u}, \vec{v} \rangle, \quad fv = \langle \vec{f}, \vec{v} \rangle.$$

Satz 1.30. *Die Bilinearform $a(\cdot,\cdot)$ der Variationsgleichung, die dem Ritz-Galerkin-Verfahren (1.30) zugrunde liegt, sei strikt X-elliptisch mit*

$$\exists \gamma > 0 : a(v, v) \geq \gamma \, ||v||_X^2 \quad \forall v \in X.$$

Dann ist die Koeffizientenmatrix A des hierzu äquivalenten linearen Gleichungssystems (1.33) positiv definit und damit regulär. Das bedeutet, es existiert genau eine Lösung $\vec{u} \in \mathbf{R}^n$ von (1.33).

Beweis. Sei $\vec{v} \in \mathbf{R}^n \neq 0$ beliebig. Dann gilt

$$\langle A\vec{v}, \vec{v} \rangle = a(v, v) \geq \gamma \, ||v||_X^2 > 0.$$

□

Bemerkung 1.31. *Die Funktionen, aus denen die Lösung u^n zusammengesetzt wird und die in der ersten Komponente von $a(\cdot,\cdot)$ stehen, werden* Ansatzfunktionen *genannt, in Abgrenzung zu den* Testfunktionen *v, die in der zweiten Komponente von $a(\cdot,\cdot)$ stehen. Beim Ritz-Galerkin-Verfahren, wie es hier definiert wurde, ist der Raum der Ansatzfunktionen gleich dem der Testfunktionen. Außerdem sind beide Räume Teilräume von X, weshalb man von einer* konformen *Methode spricht.*

Als spezielles Ritz-Galerkin-Verfahren betrachtet man nun die *Methode der finiten Elemente (FEM)*. Sie ist dadurch charakterisiert, daß man das Gebiet Ω in Teilgebiete $T_i, i = 1,\dots,m$ zerlegt, die eine einfache geometrische Struktur haben. Diesen Vorgang bezeichnet man auch als *Gitter-* oder *Netz-Generierung* (engl. *grid-generation* bzw. *mesh-generation*). Des weiteren definiert man die Test- und Ansatzfunktionen lokal über diesen Teilgebieten, so daß sie einen kleinen Träger aufweisen. Deshalb hat die Matrix A nur wenig von Null verschiedene Einträge, sie ist *dünnbesetzt*.
Typischerweise wählt man eine Zerlegung in Dreiecke (im Fall $n = 2$, hierbei spricht man auch von einer Triangulierung), bzw. Tetraeder (im Fall $n = 3$), so daß

$$\overline{\Omega} = \bigcup_{i=1}^{m} \overline{T}_i \quad \text{mit } T_j \cap T_i = \emptyset,\ j \neq i.$$

Nun stellt sich noch die Frage, welchen endlich-dimensionalen Teilraum man für die Approximation benutzt. Schließlich hängt die Güte, also der Fehler $||u - u^n||_X$ entscheidend davon ab. Bei den Finite-Elemente-Methoden verwendet man meistens Polynome vom Grad k, auf den einzelnen Teilgebieten also etwa

$$X_n = \bigcup_{i=1}^{m} \mathbf{P}_k(T_i)$$

mit

$$\mathbf{P}_k(T_i) := \{v \in W^{1,2}(\overline{\Omega}) \text{ mit } v|_{T_i} \in \Pi_k(T_i) \text{ und } v(x) = 0 \text{ für } x \in \overline{\Omega} \setminus T_i\}.$$

Häufig begnügt man sich mit Elementen vom $\mathbf{P}_1$-Typ, d.h. stückweise linearen Funktionen.

1.3.2 Stabilisierte Verfahren : Streamline Diffusion

Wie bereits in Bemerkung (1.27) erwähnt wurde, beruht die Elliptizität von $a(\cdot,\cdot)$ entscheidend auf der Tatsache, daß $\varepsilon > 0$ ist. Für $\varepsilon \to 0$ gehen somit wesentliche Voraussetzungen für die Lösbarkeit verloren. In solchen Fällen treten *Grenzschichten* auf und die numerische Lösung oszilliert stark (siehe z.B. [RST96]) so daß das Ergebnis unbrauchbar wird.

Als Ausweg versucht man die Variationsgleichung aus (1.28) zu stabilisieren, indem man künstliche Diffusion in Strömungsrichtung einführt. Dies ist die Idee bei der *Streamline-Diffusion-Finite-Element-Method (SDFEM)*, die im folgenden vorgestellt wird.

Die Bilinearform $a_{SD}(\cdot,\cdot)$

Man betrachtet hier vereinfachend eine Triangulierung $\mathcal{T}_h = (T_i)_{i=1}^m$ des Gebiets Ω. Hierbei ist h wie üblich eine Obergrenze für die Höhe der Dreiecke, also ein Maß für die Feinheit

des Gitters. Bei SDFEM werden zusätzlich zur normalen Ritz-Galerkin-Finite-Elemente-Methode gewichtete Residuen hinzuaddiert. Sei im folgenden wieder $X_n \subset X$ ein endlichdimensionaler Unterraum von $X = W^{1,2}(\Omega)$, mit $\dim X_n = n < \infty$. Nimmt man an, daß die Lösung u des unstabilisierten Problems (1.28) regulär ist, in dem Sinne, daß gilt :

$$-\varepsilon\Delta u + \vec{b}\cdot\nabla u + cu = f \quad \text{in} \quad L^2(T_i),$$

auf jedem Dreieck T_i, dann genügt u der Gleichung

$$a_{SD}(u, v_h) = f_{SD}(v_h) \quad \forall v_h \in X_n, \tag{1.34}$$

wobei $a_{SD}(\cdot,\cdot)$ und $f_{SD}(\cdot)$ wie folgt definiert sind :

$$a_{SD}(u, v) := a(u, v) + \sum_{i=1}^{m} \delta_{T_i}(-\varepsilon\Delta u + b\cdot\nabla u + cu, \vec{b}\cdot\nabla v)_{T_i}, \tag{1.35}$$

$$f_{SD} := f(v) + \sum_{i=1}^{m} \delta_{T_i}(f, \vec{b}\cdot\nabla v)_{T_i}. \tag{1.36}$$

Hierbei ist $(\cdot,\cdot)_{T_i}$ das Skalarprodukt im Hilbert-Raum $L^2(T_i)$ und $a(u, v)$ und $f(v)$ sind wie in (1.26) bzw. (1.27) definiert.

Die Konstanten δ_{T_i}, genannt SD-Parameter, sind frei wählbar, jedoch sollten sie praktischerweise so gewählt werden, daß sie eine geeignete Fehlernorm minimieren. Auf die optimale Wahl dieser Parameter wird weiter unten eingegangen.

Im allgemeinen ist $\Delta u_h \notin L^2(\Omega)$, da jedoch für jedes T_i $\Delta u_h \in L^2(T_i)$ ist, kann man Δu_h Element für Element berechnen. Für $\mathbf{P}_1$-Elemente verschwindet Δu_h.

Definition 1.32 (SDFEM). *Seien $a_{SD}(\cdot,\cdot)$ und $f_{SD}(\cdot)$ wie in (1.35) bzw. (1.36) definiert. Dann bezeichnet man die diskrete Fragestellung*

$$\begin{aligned} &\textit{Finde } u_h \in X_n, \textit{ so daß für alle } v_h \in X_n \textit{ gilt :} \\ &a_{SD}(u_h, v_h) = f_{SD}(v_h). \end{aligned} \tag{1.37}$$

als Streamline-Diffusion-Finite-Element-Method (SDFEM) *oder auch als* Streamline-Upwind-Petrov-Galerkin-Method (SUPG).

Man erhält aus (1.34) und (1.37) für $u \in W^{1,2}(\Omega)$ die Beziehung :

$$a_{SD}(u - u_h, v_h) = 0 \quad \forall v_h \in X_n. \tag{1.38}$$

Diese Orthogonalitätsaussage ist die Projektionseigenschaft von SDFEM. Eine Finite-Elemente-Methode mit dieser Eigenschaft heißt konsistent.

Für die kommenden Fehlerabschätzungen ist es üblich, die folgende Norm zu verwenden (die Normeigenschaften prüft man leicht nach) :

$$|||v|||_{SD} := \left(\varepsilon\,|v|^2_{W^{1,2}(\Omega)} + \sum_{i=1}^{m} \delta_{T_i}||\vec{b}\cdot\nabla v||^2_{L^2(T_i)} + c_0||v||^2_{L^2(T_i)}\right)^{1/2}.$$

Koerzitivität von $a_{SD}(\cdot,\cdot)$

Um Aussagen zur Stabilität und Konvergenzanalyse herzuleiten, stellt man folgende Anforderungen an die Konstanten c_T und c_0 :

$$c_{T_i} := \max_{x\in T_i} |c(x)| \quad \text{für jedes} \quad T_i \in \mathcal{T}_h, \qquad c - \frac{1}{2}\nabla\cdot\vec{b} \geq c_0 > 0 \quad \text{auf } \Omega.$$

Außerdem benötigt man eine lokale inverse Ungleichung der Art

$$||\Delta v_h||_{L^2(T_i)} \leq \mu_{inv} h_{T_i}^{-1} |v_h|_{W^{1,2}(T_i)} \quad \forall v_h \in X_n, \tag{1.39}$$

wobei die Konstante μ_{inv} unabhängig von T_i und h ist, siehe auch [RST96], Seite 230.

Satz 1.33. *Der SD-Parameter δ_{T_i} genüge*

$$0 < \delta_{T_i} \leq \frac{1}{2}\min\Big(\frac{c_0}{c_{T_i}^2}, \frac{h_{T_i}^2}{\varepsilon\mu_{inv}^2}\Big) \tag{1.40}$$

für jedes $T_i \in \mathcal{T}_h$. Dann ist die diskrete Bilinearform $a_{SD}(\cdot,\cdot)$ strikt koerzitiv. Und zwar gilt :

$$a_{SD}(v_h, v_h) \geq \frac{1}{2}|||v_h|||_{SD}^2 \qquad \forall v_h \in X_n.$$

Beweis. Es gilt für $v_h \in X_n$:

$$\begin{aligned} a_{SD}(v_h, v_h) &= \varepsilon(\nabla v_h, \nabla v_h)_\Omega + (\vec{b}\cdot\nabla v_h, v_h)_\Omega + (cv_h, v_h)_\Omega \\ &\quad + \sum_{i=1}^{m} \delta_{T_i}(-\varepsilon\Delta v_h + \vec{b}\cdot\nabla v_h + cv_h, \vec{b}\cdot\nabla v_h)_{T_i}. \end{aligned}$$

Man betrachtet nun die einzelnen Bestandteile der Gleichung, zunächst den Diffusionsterm:

$$\begin{aligned} \varepsilon(\nabla v_h, \nabla v_h)_\Omega &= \varepsilon \int_\Omega \sum_{i=1}^{n} \Big(\frac{\partial v_h}{\partial x_i}\Big)^2 dx \\ &= \varepsilon\, |v_h|_{W^{1,2}(\Omega)}^2 . \end{aligned}$$

Durch partielle Integration erhält man, da das Randintegral wegen $v_h \in W_0^{1,2}(\Omega)$ verschwindet :

$$(\vec{b}\cdot\nabla v_h, v_h)_\Omega = \sum_{i=1}^{n} \int_\Omega b_i \frac{\partial v_h}{\partial x_i} v_h dx = \sum_{i=1}^{n} -\int_\Omega \Big(\frac{\partial v_h}{\partial x_i} b_i + \frac{\partial b_i}{\partial x_i} v_h\Big) v_h dx$$

$$\Longrightarrow \quad (\vec{b}\cdot\nabla v_h, v_h)_\Omega = \int_\Omega \sum_{i=1}^{n} b_i \frac{\partial v_h}{\partial x_i} v_h dx = -\frac{1}{2}\int_\Omega \sum_{i=1}^{n} \frac{\partial b_i}{\partial x_i} v_h^2 dx = (-\frac{1}{2}(\nabla\cdot\vec{b})v_h, v_h)_\Omega$$

$$\Longrightarrow \quad (\vec{b}\cdot\nabla v_h, v_h) + (cv_h, v_h) = \big(\underbrace{(c - \frac{1}{2}(\nabla\cdot\vec{b}))}_{\geq c_0} v_h, v_h\big) \geq (c_0 v_h, v_h) = c_0\, ||v_h||_{L^2(\Omega)}^2 .$$

Deshalb ist :

$$\begin{aligned} a_{SD}(v_h, v_h) \geq \varepsilon \, |v_h|^2_{W^{1,2}(\Omega)} + c_0 \, ||v_h||^2_{L^2(\Omega)} + \sum_{i=1}^{m} \delta_{T_i} ||\vec{b} \cdot \nabla v_h||^2_{L^2(T_i)} \\ + \sum_{i=1}^{m} \delta_{T_i} (-\varepsilon \Delta v_h + c v_h, \vec{b} \cdot \nabla v_h)_{T_i}. \end{aligned} \tag{1.41}$$

Als nächstes betrachtet man :

$$\begin{aligned} \left| \sum_{i=1}^{m} \delta_{T_i} (-\varepsilon \Delta v_h + c v_h, b \cdot \nabla v_h)_{T_i} \right| &\leq \sum_{T_i} \int_{T_i} \left| \delta_{T_i}^{\frac{1}{2}} (-\varepsilon \Delta v_h + c v_h) \, \delta_{T_i}^{\frac{1}{2}} (\vec{b} \cdot \nabla v_h) \right| dx \\ &\underset{Young}{\leq} \sum_{T_i} \int_{T_i} \frac{1}{2} \delta_{T_i} (-\varepsilon \Delta v_h + c v_h)^2 + \frac{1}{2} \delta_{T_i} (\vec{b} \cdot \nabla v_h)^2 dx \\ &= \sum_{T_i} \delta_{T_i} \frac{1}{2} || - \varepsilon \Delta v_h + c v_h||^2_{L^2(T_i)} + \delta_{T_i} \frac{1}{2} ||\vec{b} \cdot \nabla v_h||^2_{L^2(T_i)} \\ &\underset{Young}{\leq} \sum_{T_i} \varepsilon^2 \delta_{T_i} ||\Delta v_h||^2_{L^2(T_i)} + \sum_{T_i} c^2_{T_i} \delta_{T_i} ||v_h||^2_{L^2(T_i)} \\ &\quad + \frac{1}{2} \sum_{T_i} \delta_{T_i} ||\vec{b} \cdot \nabla v_h||^2_{L^2(T_i)}. \end{aligned}$$

Anwenden der inversen Ungleichung (1.39) ergibt zusammen mit der vorausgesetzten Eigenschaft (1.40) :

$$\left| \sum_{i=1}^{n} \delta_{T_i} (-\varepsilon \Delta v_h + c v_h, b \cdot \nabla v_h)_{T_i} \right| \leq \frac{\varepsilon}{2} \, |v_h|^2_{W^{1,2}(\Omega)} + \frac{c_0}{2} \, ||v_h||^2_{L^2(\Omega)} + \frac{1}{2} \sum_{T_i} \delta_{T_i} ||\vec{b} \cdot \nabla v_h||^2_{L^2(T_i)}.$$

Zusammen mit (1.41) ergibt dies die Behauptung. □

Die Lax-Milgram-Theorie liefert nun die Existenz und Eindeutigkeit einer Lösung von (1.37).

Da die Lösung u_h von (1.34) eine andere ist als die der ursprünglichen Variationsgleichung, ist es noch wichtig, zu wissen, wie „weit“ u_h und die beste Approximation u^n von X_n an u auseinanderliegen. Ein Ergebnis wird durch folgenden Satz ausgedrückt :

Satz 1.34. *Für u_h und u^I gilt :*

$$|||u^n - u_h|||_{SD} \leq C h^k \big(\sum_{T_i} (\varepsilon + \delta_{T_i} + \delta_{T_i}^{-1} h^2_{T_i} + h^2_{T_i}) |u|^2_{k+1,T_i} \big)^{1/2}. \tag{1.42}$$

Beweis. Vgl. [RST96], Seite 232 □

Die Wahl der SD-Parameter δ_{T_i}

Um die rechte Seite der Gleichung (1.42) möglichst klein zu halten und dadurch die beste Konvergenzgeschwindigkeit zu erhalten, ist es erforderlich, die Terme $\varepsilon, \delta_{T_i}$ und $\delta_{T_i}^{-1} h_T^2$ auszubalancieren, und zwar unter der Nebenbedingung für δ_{T_i} aus Satz 1.33. In [RST96], Seite 233, wird hierfür folgendes vorgeschlagen :

$$\delta_{T_i} = \begin{cases} \delta_0 h_{T_i} & \text{falls } Pe_{T_i} > 1 \\ \delta_1 h_{T_i}^2 / \varepsilon & \text{falls } Pe_{T_i} \leq 1 \end{cases} \tag{1.43}$$

mit geeigneten Konstanten δ_0 und δ_1. Hierbei ist Pe_{T_i} die *Peclet-Zahl* :

$$Pe_{T_i} := \frac{b_{T_i} h_{T_i}}{\varepsilon}$$

und $b_{T_i} := \max\limits_{i=1,\ldots,n} ||b_i||_{L^\infty(T_i)}$. Die Peclet-Zahl ist eine Art Indikator dafür, ob das Problem konvektionsdominiert ($Pe_{T_i} > 1$) oder diffusionsdominiert ($Pe_{T_i} \leq 1$) ist. Oft reicht es, künstliche Diffusion nur dort zu addieren, wo $Pe_{T_i} > 1$ gilt, so daß man z.B. $\delta_0 = \frac{1}{2b_{T_i}}$ und $\delta_1 = 0$ wählt (vgl. [Kno99] Seite 70). Das selbe asymptotische Verhalten erreicht man jedoch auch mit der Formel

$$\delta_{T_i} = \frac{h_{T_i}^2}{2\varepsilon} \left(1 + Pe_{T_i}^2\right)^{-\frac{1}{2}}, \tag{1.44}$$

wie sie in den FEM-Programmen $\mathcal{PNS}$ (vgl. [AO+99]) und **adr** (siehe Kapitel 4) implementiert ist.

Eine physikalische Interpretation für SDFEM

Als Motivation für SDFEM kann man den folgenden (zweidimensionalen) Spezialfall betrachten. Es werden nur stückweise lineare Elemente verwendet, also $v_h|_{T_i} \in P_1(T_i)$. Weiterhin nimmt man an, daß $\vec{b} \equiv (b_1, b_2)$ konstant ist und $c \equiv 0$. In diesem Fall vereinfacht sich die Bilinearform $a_{SD}(\cdot,\cdot)$ zu

$$a_{SD}(u_h, v_h) = \varepsilon(\nabla u_h, \nabla v_h)_\Omega + (\vec{b} \cdot \nabla u_h, v_h)_\Omega + \sum_{T_i} \delta_{T_i} (\vec{b} \cdot \nabla u_h, \vec{b} \cdot \nabla v_h)_{T_i}.$$

Zieht man nun die Euklidische Norm ($||\cdot||_2$) von $\vec{b}$ aus beiden Komponenten des Skalarproduktes $(\vec{b} \cdot \nabla u_h, \vec{b} \cdot \nabla v_h)_{T_i}$ heraus, so kann man $\vec{b} \cdot \nabla u_h$ und $\vec{b} \cdot \nabla v_h$ als Richtungsableitungen auffassen, und zwar in Richtung der Strömung, die ja den Geschwindigkeitvektor $\vec{b}$ hat. Dann ergibt sich :

$$a_{SD}(u_h, v_h) = \varepsilon(\nabla u_h, \nabla v_h)_\Omega + (\vec{b} \cdot \nabla u_h, v_h)_\Omega + \sum_{T_i} \delta_{T_i} ||\vec{b}||_2^2 \left(\frac{\partial u_h}{\partial \vec{b}}, \frac{\partial u_h}{\partial \vec{b}}\right)_{T_i}.$$

Zusätzlich zum eigentlichen Diffusionsterm $\varepsilon(\nabla u_h, \nabla v_h)_\Omega$ hat man bei SDFEM also noch künstliche Diffusion in Höhe von $\delta_{T_i} ||\vec{b}||_2^2$ eingeführt, jedoch nur in Stromlinienrichtung - daher der Name *Streamline Diffusion*. Hierdurch wird bei kleinem Parameter ε der positive definite Anteil der Bilinearform gestärkt - hierin besteht also die Stabilisierung. Über

die Wahl der Parameter δ_{T_i} hat man dann noch zusätzlichen Einfluß auf diese künstlich eingeführte Diffusion bzw. die Ableitungen in Strömungsrichtung. Für $\delta_{T_i} = 0$ ergibt sich wieder die klassische Finite-Elemente-Methode.

Als abschließende Bemerkung läßt sich feststellen, daß man SDFEM auch als Petrov-Galerkin-Verfahren betrachten kann. Nimmt man nämlich als Testfunktionen $v + \delta_{T_i} \vec{b} \cdot \nabla v$ auf jedem $T_i \in \mathcal{T}_h$ (also Testfunktionen ungleich Ansatzfunktionen), so kommt man wieder auf die Gleichungen (1.35) bis (1.37). Aus diesem Grunde (vgl. [RST96], Seite 234) bezeichnet man SDFEM auch als Streamline Upwind Petrov Galerkin Method (SUPG).

Kapitel 2

Lösungsverfahren für lineare Gleichungssysteme

Wie man im vorigen Kapitel gesehen hat, gelangt man vom Ausgangsproblem (1.20) über die Diskretisierung zu einem linearen Gleichungssystem. Die Bestimmung einer Lösung $x \in \mathbf{R}^n$ eines solchen Systems

$$Ax = b \tag{2.1}$$

mit der Koeffizientenmatrix $A \in \mathbf{R}^{n \times n}$, der rechten Seite $b \in \mathbf{R}^n$ und der Dimension $n \in \mathbf{N}$ steht nun im Mittelpunkt der weiteren Untersuchungen. Es wird stets vorausgesetzt, daß A nichtsingulär ist und somit eine eindeutige Lösung x existiert.
In diesem Kapitel wird hierbei nicht auf traditionelle direkte Lösungsverfahren wie z.B. das Gaußsche Eliminationsverfahren (GEV) eingegangen, da sie bei der gängigen Größenordnung von n viel zu uneffizient wären (das GEV hat einen Aufwand, der bei $O(n^3)$ Operationen liegt). Statt dessen verwendet man beim Lösen solcher Probleme Iterationsverfahren, die eine Näherungslösung $x^{(k)}$ an die exakte Lösung x liefern. Man versucht hierbei in der Regel, die spezielle Struktur von A auszunutzen. Da bei Finite-Elemente-Methoden die Ansatz- und Testfunktionen einen kleinen Träger haben, hat die Matrix A nur wenig von Null verschiedene Einträge, das heißt, sie ist *dünn-* oder *schwachbesetzt*. Entsprechende Speichertechniken und Algorithmen vorausgesetzt, ist nun beispielsweise der Aufwand einer Matrix-Vektor-Multiplikation, also der Berechnung des Produktes Ay mit $y \in \mathbf{R}^n$, nur noch proportional zu der Anzahl der Nichtnullelemente von A, statt proportional zu n^2, wie bei einer vollbesetzten Matrix. Dies kommt den meisten Iterationsverfahren zugute, in denen in jedem Schritt A oder Teilmatrizen von A mit einem Vektor multipliziert werden.

Zunächst werden in diesem Kapitel einige Begriffe eingeführt, die im Zusammenhang mit Matrizen und Lösungsverfahren von Bedeutung sind. Dann werden die traditionellen Iterationsmethoden, welche auf einer Zerlegung der Matrix A basieren, vorgestellt und unter dem Begriff *Splitting-Methoden* zusammengefaßt. Danach wird auf einige der moderneren *Krylov-Unterraummethoden* eingegangen. Zum Schluß wird untersucht, wie man die Konvergenzgeschwindigkeit dieser Verfahren durch geeignete *Vorkonditionierung* verbessern kann.

Bemerkung 2.1. *Die Algorithmen sind weitestgehend so formuliert, wie sie auch in der*

Klassenbibliothek (siehe Kapitel 4) implementiert wurden. Eine Zeile wie

$$r \Leftarrow x_{tmp} - Lx \tag{2.2}$$

bedeutet z.B: „Berechne das Produkt aus der Matrix L und dem Vektor x, ziehe das Ergebnis von dem Vektor x_{tmp} ab und speichere das Endergebnis im Vektor r“ *und entspricht auch genau einer Zeile im endgültigen Programmcode. Es wurde auf eine weitgehende Eins-zu-eins Übersetzung des C++-Codes in Pseudocode Wert gelegt, lediglich Dinge wie Speicherverwaltung, Variablendeklarationen und andere implementierungsspezifische Details wurden weggelassen.*

2.1 Grundlagen und Begriffe

Für die Beschreibung von Iterationsverfahren und deren Konvergenzanalyse sind verschiedene Normen unerläßlich. Seien im folgenden $x = (x_i)_{i=1}^n \in \mathbf{R}^n$ ein Vektor, $A = (a_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ eine symmetrische und positiv definite Matrix und $B = (b_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ eine Matrix.
Für Vektoren werden folgende Normen verwendet :

- die Euklidische Norm
$$||x||_2 := \sqrt{x^T x} = \left(\sum_{i=1}^n x_i^2\right)^{\frac{1}{2}} \tag{2.3}$$
- die L_1-Norm
$$||x||_1 := \sum_{i=1}^n |x_i| \tag{2.4}$$
- die Maximumnorm
$$||x||_\infty := \max_{i=1,\ldots,n} |x_i| \tag{2.5}$$
- die Energienorm
$$||x||_A := \sqrt{x^T A x} \tag{2.6}$$

Die einer Vektornorm $||\cdot||_*$ *zugeordnete* oder *zugehörige Matrixnorm* $||\cdot||_{*,*}$ ist bekanntlich gegeben durch

$$||B||_{*,*} := \sup_{\substack{y\in\mathbf{R}^n \\ ||y||_*=1}} ||By||_* = \sup_{\substack{y\in\mathbf{R}^n \\ ||y||_*\neq 0}} \frac{||By||_*}{||y||_*}. \tag{2.7}$$

Bevor jedoch auf die explizite Darstellung der zu den oben aufgeführten Vektornormen zugeordneten Matrixnormen eingegangen wird, noch ein paar wichtige Aussagen über Matrizen.

Definition 2.2 (Spektralradius). *Seien $\lambda_1, \ldots, \lambda_n$ die Eigenwerte der Matrix $B \in \mathbf{R}^{n\times n}$. Dann nennt man die Zahl*

$$\rho(B) := \max_{i=1,\ldots,n} |\lambda_i|$$

den Spektralradius *von B.*

Für eine symmetrisch positiv definite Matrix A existiert eine Darstellung $V^T AV = D$ mit einer orthogonalen Matrix $V \in \mathbf{R}^{n\times n}$ und einer Diagonalmatrix $D = diag(d_{11}, \ldots, d_{nn}) \in \mathbf{R}^{n\times n}$, $d_{ii} > 0$ für $i = 1, \ldots, n$. Definiert man nun

$$D^{\frac{1}{2}} := diag(\sqrt{d_{11}}, \ldots, \sqrt{d_{nn}}), \tag{2.8}$$

dann läßt sich zu A eine „Wurzelmatrix“ $A^{\frac{1}{2}}$ angeben, in der Form

$$A^{\frac{1}{2}} := V^T D^{\frac{1}{2}} V. \tag{2.9}$$

Weil $VV^T = I$ gilt, ist folgendes gezeigt :

Lemma 2.3. *Zu jeder symmetrisch positiv definiten Matrix $A \in \mathbf{R}^{n\times n}$ existiert eine (ebenfalls symmetrisch positiv definite) Matrix $A^{\frac{1}{2}}$, für die gilt*

$$A^{\frac{1}{2}} A^{\frac{1}{2}} = A.$$

Satz 2.4. *Die zu den Vektornormen (2.3) bis (2.6) zugeordneten Matrixnormen sind nun*

- *die Spektralnorm*
$$||B||_{2,2} := \sqrt{\rho\,(B^T B)} \tag{2.10}$$
- *die Spaltensummennorm*
$$||B||_{1,1} := \max_{j=1,\ldots,n} \sum_{i=1}^{n} |b_{ij}| \tag{2.11}$$
- *die Zeilensummennorm*
$$||B||_{\infty,\infty} := \max_{i=1,\ldots,n} \sum_{j=1}^{n} |b_{ij}| \tag{2.12}$$
- *die Energienorm*
$$||B||_{A,A} := \left|\left| A^{\frac{1}{2}} B A^{-\frac{1}{2}} \right|\right|_{2,2} \tag{2.13}$$

Beweis. Ergibt sich durch Nachrechnen der Normaxiome und durch Nachprüfen der Eigenschaft (2.7). □

2.2 Splitting-Methoden

Die elementaren Iterationsverfahren beruhen auf einer Zerlegung von A in eine Matrix $N \in \mathbf{R}^{n\times n}$ und eine invertierbare Matrix $M \in \mathbf{R}^{n\times n}$ mit der Eigenschaft $A = M - N$. Dadurch läßt sich Gleichung (2.1) überführen in $Mx = Nx + b$. Dies ist die Motivation für die Iterationsformulierung $Mx^{(k+1)} = Nx^{(k)} + b$ und damit

$$x^{(k+1)} := M^{-1} N x^{(k)} + M^{-1} b. \tag{2.14}$$

Die Matrix $C := M^{-1}N$ heißt *Iterationsmatrix* zur Iterationsvorschrift (2.14).
Für die praktische Anwendung ist es wichtig, daß M leicht zu invertieren ist. Bei den vorgestellten klassischen Verfahren wird dies durch die Verwendung von Diagonal- oder

Dreiecksmatrizen bzw. daraus gebildeten Produkten erreicht.

Nun stellt sich die Frage, unter welchen Voraussetzungen das durch (2.14) definierte Verfahren gegen die Lösung x konvergiert. Hierzu läßt sich eine allgemeine Aussage machen, die nicht nur ein hinreichendes, sondern auch ein notwendiges Kriterium formuliert.

Satz 2.5. *Sei $A = M - N \in \mathbf{R}^{n\times n}$ eine Zerlegung von A mit nichtsingulärem $M \in \mathbf{R}^{n\times n}$ und $b \in \mathbf{R}^n$ gegeben. Dann sind folgende zwei Aussagen äquivalent :*

(i.) Für den Spektralradius von $M^{-1}N$ gilt $\rho(M^{-1}N) < 1$.

(ii.) A ist nichtsingulär, und die aus der Vorschrift

$$x^{(k+1)} := M^{-1}Nx^{(k)} + M^{-1}b$$

gebildete Folge $\{x^{(k)}\}$ konvergiert für jedes $x^{(0)} \in \mathbf{R}^n$ gegen die eindeutige Lösung x^* der Gleichung $Ax = b$.

Beweis. Siehe z.B. [Dem97] S. 280, [Saa96] S. 104 oder [Wer92] S. 123. □

Die nun dargestellten Verfahren wie Jacobi, Gauß-Seidel oder SOR verwenden alle eine Aufteilung von A in der Form

$$A = L + D + R, \tag{2.15}$$

wobei diese Teilmatrizen wie folgt definiert sind :

- $L = (l_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ ist die strikte, linke untere Teildreiecksmatrix von A, d.h.

$$l_{ij} := \begin{cases} a_{ij} & \text{für } i \geq j+1 \\ 0 & \text{sonst} \end{cases}$$

- $D = (d_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ ist die Matrix, die aus den Diagonalelementen von A gebildet wird, d.h.

$$d_{ij} := \begin{cases} a_{ij} & \text{für } i = j \\ 0 & \text{sonst} \end{cases}$$

- $R = (r_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ ist die aus dem strikten, rechten oberen Teil von A gebildete Dreiecksmatrix, also

$$r_{ij} := \begin{cases} a_{ij} & \text{für } i \leq j+1 \\ 0 & \text{sonst} \end{cases}$$

Es wird angenommen, daß D nichtsingulär ist, was gleichbedeutend damit ist, daß keines der Diagonalelemente von A verschwindet. Dies läßt sich, wenn A nichtsingulär ist, immer durch Umordnen der Zeilen oder Spalten erreichen.

2.2.1 Das Jacobi-Verfahren

Das einfachste Iterationsverfahren stellt das *Jacobi-Verfahren* dar, das man auch als *Gesamtschrittverfahren* bezeichnet. Hierbei ist $M = D$ und $N = -(L + R)$, und damit die Iterationsmatrix

$$C_J := -D^{-1}(L + R) = I - D^{-1}A. \tag{2.16}$$

Eine praktische Realisierung, wie sie auch in der Softwarebibliothek (siehe Kapitel 4) implementiert ist, sieht folgendermaßen aus :

Algorithmus 2.1 Jacobi

Require: D^{-1} existiert $\vee\ \delta > 0$
1: $x_{tmp} \Leftarrow b$
2: $x_{tmp} \Leftarrow x_{tmp} - Lx$
3: $x_{tmp} \Leftarrow x_{tmp} - Rx$
4: $r \Leftarrow x_{tmp} - Dx$
5: **while** $||r|| > \delta$ **do**
6: $\quad x \Leftarrow D^{-1}x_{tmp}$
7: $\quad x_{tmp} \Leftarrow b$
8: $\quad x_{tmp} \Leftarrow x_{tmp} - Lx$
9: $\quad x_{tmp} \Leftarrow x_{tmp} - Rx$
10: $\quad r \Leftarrow x_{tmp} - Dx$
11: **end while**

Bei dieser Formulierung des Algorithmus wird das *Residuum* $r = b - Ax$ im Vorbeigehen berechnet und für eine Abbruchbedingung verwendet. Wie allgemein üblich, endet das Verfahren, wenn $||r|| < \delta$ in einer gewissen Norm $||\cdot||$ mit einem $\delta \in \mathbf{R}, \delta > 0$ erfüllt ist.

Bemerkung 2.6. *In Algorithmus (2.1) wird über die Problemdaten A, b, x hinaus Speicherplatz für zwei Vektoren der Dimension n benötigt : für das Residuum r und den temporären Hilfsvektor x_{tmp}.*

Ein hinreichendes, aber einfacher nachzuprüfendes Konvergenzkriterium für das Gesamtschrittverfahren als das in Satz (2.5) gegebene liefert der folgende

Satz 2.7 (Starkes Zeilen- und Spaltensummenkriterium). *Das Jacobi-Verfahren zur Lösung des linearen Gleichungssystems $Ax = b$ mit der Koeffizientenmatrix $A = (a_{ij})_{i,j=1}^{n} \in \mathbf{R}^{n\times n}$ konvergiert für jeden Startvektor und jede rechte Seite, falls das* starke Zeilensummenkriterium

$$\sum_{\substack{k=1 \\ k\neq j}}^{n} |a_{jk}| < |a_{jj}| \quad \forall j,\ 1 \le j \le n \tag{2.17}$$

oder das starke Spaltensummenkriterium

$$\sum_{\substack{j=1 \\ j\neq j}}^{n} |a_{jk}| < |a_{kk}| \quad \forall k,\ 1 \le k \le n \tag{2.18}$$

erfüllt ist.

Beweis. Für die Einträge der Iterationsmatrix $C_J = (c_{ij})_{i,j=1}^n$ gilt

$$c_{ij} = \begin{cases} \frac{a_{ij}}{a_{ii}} & \text{für } i \neq j \\ 0 & \text{sonst} \end{cases}$$

d.h. das starke Zeilensummenkriterium ist äquivalent zu $||C_J||_{\infty,\infty} < 1$. Entsprechend ist das Spaltensummenkriterium äquivalent zu $||C_J||_{1,1} < 1$. Es ergibt sich, da diese Matrixnormen passend zu $||\cdot||_\infty$ bzw. $||\cdot||_1$ sind : $||C_J y - C_J z|| < ||C_J|| \cdot ||y - z||$ für alle $y, z \in \mathbf{R}^n$. Man hat also eine stark kontrahierende Abbildung vorliegen, und die Anwendung des Banachschen Fixpunktsatzes liefert die Konvergenz des Verfahrens. □

Matrizen, die das starke Zeilensummenkriterium erfüllen, heißen *strikt diagonaldominant.*

2.2.2 Das Gauß-Seidel-Verfahren

Beim Gauß-Seidel-Verfahren hat man die Zerlegung $M = L + D$ und $N = -R$ vorliegen. Dies ergibt die Iterationsgleichung $(L + D)x^{(k+1)} = -Rx^{(k)} + b$, was einem linearen Gleichungssystem mit einer Koeffizientenmatrix von linker unterer Dreiecksgestalt entspricht. Die Iterationsmatrix hat die Form

$$C_{GS} := -(L + D)^{-1} R. \tag{2.19}$$

Das explizite Ausrechnen von $(L + D)^{-1}$ wäre in der Praxis allerdings viel zu aufwendig. Statt dessen behilft man sich, indem man in jedem Schritt die Komponenten des Vektors $x^{(k+1)} = (x_1^{(k+1)}, \ldots, x_n^{(k+1)})^T$ sukzessive von $i = 1$ bis n durch „Vorwärtseinsetzen“ berechnet. Ist nnz_L die Anzahl der Nichtnullelemente in L, dann beträgt der Aufwand hierfür $n + nnz_L$ Operationen.

Nachfolgend wird eine praxisorientierte Darstellung des Verfahrens gegeben, das in der Literatur auch unter dem Namen *Einzelschrittverfahren* geführt wird. Auch hierbei wird wieder Speicherplatz für das zur Konvergenzkontrolle genutzte Residuum r und für den Hilfsvektor x_{tmp} benötigt.

Algorithmus 2.2 Gauß-Seidel

Require: $\delta > 0$
1: $x_{tmp} \Leftarrow b$
2: $x_{tmp} \Leftarrow x_{tmp} - Rx$
3: $r \Leftarrow x_{tmp} - Lx$
4: $r \Leftarrow r - Dx$
5: **while** $||r|| > \delta$ **do**
6: Subroutine : Bestimme x aus $(L + D)x = x_{tmp}$ durch Vorwärtseinsetzen.
7: $x_{tmp} \Leftarrow b$
8: $x_{tmp} \Leftarrow x_{tmp} - Rx$
9: $r \Leftarrow x_{tmp} - Lx$
10: $r \Leftarrow r - Dx$
11: **end while**

Da die Matrix $L + D$ eine bessere Näherung an A ist als D, erwartet man auch, daß $(L + D)^{-1}$ die Inverse von A besser approximiert als D^{-1} und demzufolge, daß das Gauß-Seidel-Verfahren besser konvergiert als das Jacobi-Verfahren. Diese Vermutung bestätigt der folgende Satz.

Satz 2.8 (Sassenfeld-Kriterium). *Definiert man rekursiv*

$$s_i := \sum_{j=1}^{i-1} \left| \frac{a_{ij}}{a_{ii}} \right| s_j + \sum_{j=i+1}^{n} \left| \frac{a_{ij}}{a_{ii}} \right|, \quad 1 \leq i \leq n,$$

dann gilt für die Iterationsmatrix C_{GS} des Gauß-Seidel-Verfahrens aus (2.19)

$$||C_{GS}||_{\infty,\infty} \leq \max_{1 \leq i \leq n} s_i =: s.$$

Für $s < 1$ konvergiert deshalb das Verfahren für jeden Startvektor und jede rechte Seite. Ist zusätzlich das starke Zeilensummenkriterium erfüllt, so gilt

$$||C_{GS}||_{\infty,\infty} \leq s \leq ||C_J||_{\infty,\infty} < 1.$$

Beweis. Siehe z.B. [Die96], Satz 6.2 Seite 143. □

In diesem Fall kann man also eine etwas bessere Konvergenzrate als beim Jacobi-Verfahren erwarten.
Vertauscht man im Gauß-Seidel-Verfahren die Teilmatrizen L und R, so führt dies auf die Zerlegung $M = D + R$ und $N = -L$. Man erhält die Iterationsgleichung

$$(D + R)x^{(k+1)} = -Lx^{(k)} + b, \tag{2.20}$$

ein lineares Gleichungssystem mit rechter oberer Koeffizientenmatrix, welches man durch „Rückwärtseinsetzen“ löst, also durch sukzessives Berechnen der Komponenten des Vektors $x^{(k+1)}$ in der Reihenfolge $n, n-1, \ldots, 1$. Die Iterationsvorschrift (2.20) nennt man auch einen *Rückwärts-Gauß-Seidel-Schritt.*

Kombiniert man nun einen normalen Vorwärts- mit einem Rückwärts-Gauß-Seidel-Schritt, so ergibt sich das *symmetrische Gauß-Seidel-Verfahren* :

$$(L + D)x^{(k+\frac{1}{2})} = -Rx^{(k)} + b \tag{2.21}$$

$$(D + R)x^{(k+1)} = -Lx^{(k+\frac{1}{2})} + b \tag{2.22}$$

Hierbei ist $M = (L + D)D^{-1}(D + R)$ und $N = LD^{-1}R$. Die Iterationsmatrix lautet

$$C_{SGS} := (D + R)^{-1}L(L + D)^{-1}R.$$

In Algorithmus 2.3 wird dieses Verfahren im einzelnen dargestellt.

Algorithmus 2.3 Symmetrischer Gauß-Seidel

Require: $\delta > 0$

1: $x_{tmp} \Leftarrow b$
2: $x_{tmp} \Leftarrow x_{tmp} - Rx$
3: $r \Leftarrow x_{tmp} - Lx$
4: $r \Leftarrow r - Dx$
5: **while** $||r|| > \delta$ **do**
6: Subroutine : Bestimme x aus $(L + D)x = x_{tmp}$ durch Vorwärtseinsetzen.
7: $x_{tmp} \Leftarrow b$
8: $x_{tmp} \Leftarrow x_{tmp} - Lx$
9: Subroutine : Bestimme x aus $(D + R)x = x_{tmp}$ durch Rückwärtseinsetzen.
10: $x_{tmp} \Leftarrow b$
11: $x_{tmp} \Leftarrow x_{tmp} - Rx$
12: $r \Leftarrow x_{tmp} - Lx$
13: $r \Leftarrow r - Dx$
14: **end while**

2.2.3 Das JOR-Verfahren

Die Konvergenzgeschwindigkeit der bisher vorgestellten Verfahren hängt entscheidend von der Größenordnung von $\rho(C)$ bzw. $||C||$ ab. Oft hat man aber den Fall, daß der Spektralradius von C nah bei 1 liegt, und damit eine schlechte Konvergenz.
Eine gebräuchliche Beschleunigungsstrategie ist die *Relaxation* dieser Verfahren. Ausgangspunkt ist jedesmal das skalierte Gleichungssystem

$$\omega Ax = \omega(L + D + R)x = \omega b \quad \text{mit } \omega \in \mathbf{R}. \tag{2.23}$$

Dies läßt sich zum Beispiel umformen zu $Dx = (D - \omega A)x + \omega b$, was die Motivation für das *gedämpfte* bzw. *relaxierte Jacobi-Verfahren*,

$$x^{(k+1)} := x^{(k)} - \omega D^{-1}Ax^{(k)} + \omega D^{-1}b, \tag{2.24}$$

ist, daß auch als *JOR-Verfahren* (*„jacobi overrelaxation“*) bezeichnet wird. Jedoch spricht man hier, wie bei allen relaxierten Verfahren, nur bei einem Parameter $\omega > 1$ von Überrelaxation, entsprechend bei $\omega < 1$ von Unterrelaxation oder Dämpfung. Aus 2.24 ergibt sich $M = \frac{1}{\omega}D$ und $N = \frac{1}{\omega}D - A$ und damit die Iterationsmatrix dieses Verfahrens :

$$C_{JOR}(\omega) := I - \omega D^{-1}A. \tag{2.25}$$

Algorithmus 2.4 JOR

Require: D^{-1} existiert $\vee\ \delta > 0$

1: $r \Leftarrow b$
2: $r \Leftarrow r - Ax$
3: **while** $||r|| > \delta$ **do**
4: $\quad x_{tmp} \Leftarrow \omega r$
5: $\quad x \Leftarrow x + D^{-1}x_{tmp}$
6: $\quad r \Leftarrow b$
7: $\quad r \Leftarrow r - Ax$
8: **end while**

Satz 2.9 (Konvergenz des JOR). *Sei A symmetrisch und positiv definit und μ der kleinste Eigenwert der Iterationsmatrix C_J des Jacobi-Verfahrens. Dann konvergiert das JOR-Verfahren genau dann, wenn für den Relaxationsparameter gilt $0 < \omega < \frac{2}{(1-\mu)}$.*

Beweis. Siehe z.B. Korollar 2.6, Seite 12, in [Xie95] oder Satz 4.4.14, Seite 94, in [Hac93]. □

Der optimale Relaxationsparameter läßt sich unter gewissen Bedingungen angeben, beispielsweise, wenn die Iterationsmatrix C_J des Jacobi-Verfahrens nur reelle Eigenwerte hat.

Satz 2.10. *Die Iterationsmatrix C_J, wie in (2.16) definiert, habe nur reelle Eigenwerte, worunter μ der kleinste sei. Weiterhin gelte $\rho(C_J) < 1$, dann wird der Spektralradius $\rho\big(C_{JOR}(\omega)\big)$ minimiert durch*

$$\omega_{opt} = \frac{2}{2 - \mu - \rho(C_J)}. \tag{2.26}$$

Beweis. Siehe Satz 4.20, Seite 72, in [Mei99]. □

Für den Spezialfall von *konsistent geordneten* Matrizen läßt sich auch der Spektralradius der Iterationsmatrix $C_{JOR}(\omega)$ direkt angeben.

Definition 2.11 (Konsistent geordnete Matrizen). *Eine Matrix $A \in \mathbf{R}^{n\times n}$ mit der Zerlegung (2.15) heißt* konsistent geordnet, *wenn D nichtsingulär ist und die Eigenwerte der Matrix*

$$C(\alpha) := \alpha^{-1}D^{-1}L + \alpha D^{-1}R$$

unabhängig von $\alpha \in \mathbf{C}, \alpha \neq 0$ sind.

Satz 2.12. *Zusätzlich zu den Voraussetzungen von Satz 2.10 sei A symmetrisch und konsistent geordnet. Ist ω_{opt} wie in (2.26) definiert, dann gilt*

$$\rho\big(C_{JOR}(\omega_{opt})\big) = \min_{\omega} \rho\big(C_{JOR}(\omega)\big) = \frac{\rho(C_J) - \mu}{2 - \mu - \rho(C_J)}$$

Beweis. Siehe Korollar 2.17, Seite 18, in [Xie95]. □

Bei Gleichungssystemen, deren Iterationsmatrix $C_{JOR}(\omega)$ ein symmetrisches um den Nullpunkt verteiltes Spektrum besitzt, bringt die Relaxation jedoch keine Verbesserung gegenüber dem Jacobi-Verfahren, da sich hierbei laut Satz 2.10 $\omega_{opt} = 1$ ergibt (vgl. [Mei99], Seite 73).

2.2.4 Das SOR-Verfahren

Ebenfalls aus (2.23) läßt sich auch die Gleichung $(\omega L + D)x = \omega b + ((1-\omega)D - \omega R)x$ ableiten, und daraus die Iterationsformulierung für das *relaxierte Gauß-Seidel-Verfahren*, das man überwiegend als *SOR-Verfahren* (*„successive overrelaxation“*) bezeichnet :

$$(D + \omega L)x^{(k+1)} := [(1-\omega)D - \omega R]x^{(k)} + \omega b. \tag{2.27}$$

Hierbei ist also $M = \frac{1}{\omega}(\omega L + D)$ und $N = \frac{1}{\omega}[(1-\omega)D - \omega R]$. Die Iterationsmatrix des SOR-Verfahrens hat dann die Gestalt

$$C_{SOR}(\omega) := (D + \omega L)^{-1}[(1-\omega)D - \omega R]. \tag{2.28}$$

Das Gleichungssystem (2.27) löst man wiederum durch Vorwärtseinsetzen.

Algorithmus 2.5 SOR

Require: $\delta > 0$
1: $r \Leftarrow b$
2: $r \Leftarrow r - Dx$
3: $r \Leftarrow r - Rx$
4: $x_{tmp} \Leftarrow \omega \cdot r$
5: $x_{tmp} \Leftarrow x_{tmp} + Dx$
6: $r \Leftarrow r - Lx$
7: **while** $||r|| > \delta$ **do**
8: Subroutine : Bestimme x aus $(\omega L + D)x = x_{tmp}$ durch Vorwärtseinsetzen.
9: $r \Leftarrow b$
10: $r \Leftarrow r - Dx$
11: $r \Leftarrow r - Rx$
12: $x_{tmp} \Leftarrow \omega \cdot r$
13: $x_{tmp} \Leftarrow x_{tmp} + Dx$
14: $r \Leftarrow r - Lx$
15: **end while**

Es lassen sich verschiedene Konvergenzaussagen für das SOR-Verfahren herleiten, wenn die Matrizen A bzw. $C_{SOR}(\omega)$ gewisse Eigenschaften besitzen, z.B. wenn A symmetrisch positiv definit ist.

Satz 2.13 (Kahan bzw. Ostrowski und Reich). *Sei $C_{SOR}(\omega)$ die Iterationsmatrix wie in (2.28) definiert. Dann gilt*

$$\rho\big(C_{SOR}(\omega)\big) \geq |1-\omega| \quad \textit{für } \omega \neq 0.$$

Ist A symmetrisch und positiv definit, dann gilt

$$\rho\big(C_{SOR}(\omega)\big) \leq 1 \quad \textit{für } \omega \in (0,2).$$

Beweis. Siehe z.B. [Hac93], Seite 96ff., Lemma 4.4.20 und Satz 4.4.21 oder [Die96], Seite 156ff., Satz 6.7 und Satz 6.8. □

Eine Konsequenz hieraus ist also, daß SOR für jeden Startvektor x_0 und jede rechte Seite b höchstens dann konvergiert, wenn ω Werte aus $(0,2)$ annimmt. Es stellt sich hier natürlich auch die Frage nach dem optimalen Relaxationsparameter ω_{opt}, der den Spektralradius von $C_{SOR}(\omega)$ minimiert, also

$$\rho\big(C_{SOR}(\omega_{opt})\big) = \min_{0<\omega<2} \rho\big(C_{SOR}(\omega)\big),$$

und somit die beste Konvergenzrate liefert. Für den allgemeinen Fall ist es sehr schwierig, diesen Parameter zu bestimmen, da die Iterationsmatrix $C_{SOR}(\omega)$ nichtlinear von ω abhängt. Es gelingt jedoch, wenn A konsistent geordnet ist.

In [BB85], Seite 135, wird beispielsweise gezeigt, daß $\rho(C_J)^2 = \rho(C_{GS})$ gilt, wenn A konsistent geordnet ist. Eine weitergehende Aussage ist die folgende :

Satz 2.14. *Sei A konsistent geordnet mit nichtsingulärer Diagonalmatrix D, ferner seien die Eigenwerte der Iterationsmatrix C_J des Jacobi-Verfahrens sämtlich reell und es sei*

$$\beta := \rho(C_J) < 1.$$

Dann gilt für das SOR-Verfahren :

(i.) $\omega_{opt} = \frac{2}{1+\sqrt{1-\beta^2}}$,

(ii.) $\rho\big(C_{SOR}(\omega_{opt})\big) = \frac{\beta^2}{(1+\sqrt{1-\beta^2})^2}$.

Beweis. Siehe z.B. [BB85], Seite 135ff. oder [HY81], Seite 216. □

In den meisten Fällen läßt sich jedoch ein optimaler Relaxationsparameter nur auf experimentellem Wege ermitteln.

Bemerkung 2.15. *Das Gauß-Seidel-Verfahren ist ein Spezialfall des SOR-Verfahrens mit dem Relaxationsparameter $\omega = 1$.*

2.2.5 Das SSOR-Verfahren

Beim Gauß-Seidel- wie beim SOR-Verfahren ist die Matrix M nichtsymmetrisch, auch dann, wenn A selbst symmetrisch ist. Dies macht beide Verfahren ungeeignet für die Vorkonditionierung von Verfahren, die ihrerseits die Symmetrie und positive Definitheit voraussetzen, wie z.B. das CG-Verfahren (mehr dazu in den Abschnitten 2.3.4 und 2.4.1). Deshalb sind symmetrische Splitting-Verfahren (mit symmetrischem M) von Interesse.

Ähnlich wie in Abschnitt 2.2.2 aus dem Gauß-Seidel-Verfahren das symmetrische Gauß-Seidel-Verfahren hergeleitet wurde, läßt sich nun auch eine symmetrisierte Version des SOR-Verfahrens angeben, das *SSOR-Verfahren* (*„symmetric successive overrelaxation“*). Es besteht aus einem Vorwärts- und einem Rückwärts-SOR-Schritt, analog zu (2.21) und (2.22) :

$$(D+\omega L)x^{(k+\frac{1}{2})} = ((1-\omega)D - \omega R)x^{(k)} + \omega b \tag{2.29}$$

$$(D+\omega R)x^{(k+1)} = ((1-\omega)D - \omega L)x^{(k+\frac{1}{2})} + \omega b. \tag{2.30}$$

Daraus ergeben sich $M = \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega R)$,
$N = \frac{1}{\omega(2-\omega)}[(1-\omega)D - \omega L]D^{-1}[(1-\omega)D - \omega R]$ und die Iterationsmatrix

$$C_{SSOR}(\omega) := (D+\omega R)^{-1}[(1-\omega)D - \omega L](D+\omega L)^{-1}[(1-\omega)D - \omega R].$$

Natürlich ist M nur dann symmetrisch, wenn A selbst symmetrisch ist. Die Gleichungen (2.29) bzw. (2.30) werden wieder durch Vorwärts- bzw. Rückwärtseinsetzen gelöst. Festgehalten ist all dies in Algorithmus 2.6.

Algorithmus 2.6 SSOR

Require: $\delta > 0$
1: $r \Leftarrow b$
2: $r \Leftarrow r - Dx$
3: $r \Leftarrow r - Rx$
4: $x_{tmp} \Leftarrow \omega \cdot r$
5: $x_{tmp} \Leftarrow x_{tmp} + Dx$
6: $r \Leftarrow r - Lx$
7: **while** $||r|| > \delta$ **do**
8: Subroutine : Bestimme x aus $(\omega L + D)x = x_{tmp}$ durch Vorwärtseinsetzen.
9: $x_{tmp} \Leftarrow b$
10: $x_{tmp} \Leftarrow x_{tmp} - Lx$
11: $x_{tmp} \Leftarrow \omega \cdot x_{tmp}$
12: $r \Leftarrow x - Dx$
13: $r \Leftarrow (1-\omega) \cdot r$
14: $x_{tmp} \Leftarrow x_{tmp} + r$
15: Subroutine : Bestimme x aus $(\omega R + D)x = x_{tmp}$ durch Rückwärtseinsetzen.
16: $r \Leftarrow b$
17: $r \Leftarrow r - Dx$
18: $r \Leftarrow r - Rx$
19: $x_{tmp} \Leftarrow \omega \cdot r$
20: $x_{tmp} \Leftarrow x_{tmp} + Dx$
21: $r \Leftarrow r - Lx$
22: **end while**

Bemerkung 2.16. *In den Zeilen 12 - 14 wird der Vektor r als reiner Zwischenspeicher verwendet, um den Platz für einen weiteren Hilfsvektor einzusparen. Er enthält zu diesem Zeitpunkt also nicht das Residuum.*

Für das SSOR-Verfahren läßt sich eine Konvergenzaussage treffen, für den Fall, daß A symmetrisch und positiv definit ist.

Satz 2.17. *Sei $A \in \mathbf{R}^{n\times n}$ symmetrisch und positiv definit, dann konvergiert das SSOR-Verfahren für alle $\omega \in (0,2)$, und es gilt*

$$\rho\left(C_{SSOR}(\omega)\right) = ||C_{SSOR}(\omega)||_{A,A} = ||C_{SOR}(\omega)||^2_{A,A}\,. \tag{2.31}$$

Beweis. Siehe [Mei99], Seite 89ff, Satz 4.38 oder [You71], Seite 463, Satz 2.1. □

Bemerkung 2.18. *Für diesen „gutartigen" Sonderfall hat man also auch eine Aussage zur Konvergenz des symmetrischen Gauß-Seidel-Verfahrens, da dies lediglich ein Spezialfall des SSOR-Verfahrens mit $\omega = 1$ ist.*

2.3 Projektionsmethoden und Krylov-Unterraum-Verfahren

Die in diesem Abschnitt betrachteten Methoden zur Lösung eines Gleichungssystems (2.1) basieren auf der Idee, eine Näherungslösung in einem Unterraum von $\mathbf{R}^n$ zu suchen. Ist $\mathcal{K}$ ein solcher Unterraum der Dimension m, dann werden dazu in der Regel m Bedingungen gebraucht. Üblicherweise verlangt man, daß das Residuum $b - Ax$ orthogonal zu m linear unabhängigen Vektoren ist, wodurch ein weiterer Unterraum $\mathcal{L}$ definiert wird.

Definition 2.19 (Projektionsverfahren). *Unter einer* Projektionsmethode *zur Lösung des Gleichungssystems (2.1) versteht man ein Verfahren, das eine Näherungslösung x_m aus dem affinen Unterraum $x_0 + \mathcal{K}_m$ berechnet, unter der Nebenbedingung*

$$(b - Ax_m) \perp \mathcal{L}_m. \tag{2.32}$$

Hierbei ist $x_0 \in \mathbf{R}^n$ eine beliebige Startnäherung, und $\mathcal{K}_m$ und $\mathcal{L}_m$ sind Unterräume des $\mathbf{R}^n$ mit der Dimension $m \leq n$.

Ist $\mathcal{K}_m = \mathcal{L}_m$, so steht das Residuum $r_m = b - Ax_m$ senkrecht auf $\mathcal{K}_m$. Man spricht dann von einer orthogonalen Projektionsmethode, und (2.32) heißt *Galerkin-Bedingung.*

Im Falle $\mathcal{K}_m \neq \mathcal{L}_m$ ergibt sich eine schiefe Projektionsmethode. Die Forderung (2.32) wird dann als *Petrov-Galerkin-Bedingung* bezeichnet.

Ein erster Prototyp eines Projektionsverfahrens sieht folgendermaßen aus : Hat man mit $v_1, \ldots, v_m$ eine Basis von $\mathcal{K}_m$ und mit $w_1, \ldots w_m$ eine Basis von $\mathcal{L}_m$, und seien

$$V = (v_1, \ldots, v_m) \in \mathbf{R}^{n \times m} \qquad \text{und} \qquad W = (w_1, \ldots, w_m) \in \mathbf{R}^{n \times m} \tag{2.33}$$

die aus den jeweiligen Spaltenvektoren gebildeten Matrizen, dann läßt sich die Näherungslösung x_m als Linearkombination

$$x_m = x_0 + Vy \quad \text{mit } y \in \mathbf{R}^m$$

schreiben. Bezeichnet man mit $r_0 = b - Ax_0$ das Anfangsresiduum, dann führt dies zusammen mit der Orthogonalitätsbedingung (2.32) auf die Darstellung

$$W^T AVy = W^T r_0. \tag{2.34}$$

Setzt man nun noch die Invertierbarkeit von $W^T AV$ voraus, so ergibt sich

$$x_m = x_0 + V(W^T AV)^{-1} W^T r_0 \tag{2.35}$$

als Resultat des Verfahrens.

2.3.1 Krylov-Unterräume

Man kann selbst das Gauß-Seidel-Verfahren als orthogonale Projektionsmethode auffassen (siehe z.B. [Mei99], Seite 107). Die erfolgreichsten und bekanntesten Projektionsmethoden basieren jedoch darauf, daß $\mathcal{K}_m$ als *Krylov-Unterraum* gewählt wird.

Definition 2.20 (Krylov-Unterraum). *Sei $A \in \mathbf{R}^{n\times n}$ und $v \in \mathbf{R}^n, v \neq 0$. Dann nennt man die Menge*

$$\mathcal{K}_m(A, v) := \operatorname{span}\left\{v, Av, \ldots, A^{m-1}v\right\}$$

einen Krylov-Unterraum *von* $\mathbf{R}^n$.

Bei den meisten *Krylov-Verfahren* wählt man nun $v := r_0 = b - Ax_0$ und hat damit in aller Regel $v \neq 0$ sichergestellt - andernfalls wäre x_0 ja bereits die Lösung. Dann wird eine optimale Approximation an die exakte Lösung $x = A^{-1}b$ ermittelt. Hat man nun eine solche Näherungslösung $x_m \in \mathcal{K}_m := \mathcal{K}_m(A, r_0)$ gefunden und ist mit ihr noch nicht zufrieden, so kann man entweder einen Restart machen, d.h. man setzt $x_0 := x_m$ und $r_0 := b - Ax_0$ und beginnt von vorn, oder man man erhöht die Dimension m des Suchraums $\mathcal{K}_m$. Wichtig für die Güte der Näherung an x ist hierbei natürlich die „Größe“ des Suchraums, also die Dimension des Teilraums $\mathcal{K}_m(A, r_0)$. Zu ihrer Bestimmung ist der Begriff des Grades eines Vektors hilfreich.

Definition 2.21. *Sei $A \in \mathbf{R}^{n\times n}$ und $v \in \mathbf{R}^n$. Dann nennt man*

$$\operatorname{grad}(v) := \min_{i=1,\ldots,n} \{j \mid \exists p \in \Pi_i \setminus \{0\} \; mit \; p(A)v = 0\} \tag{2.36}$$

den Grad *von* v.

Aus dem Satz von Caley-Hamilton folgt zumindest schon einmal, daß der Grad von v die Zahl n nicht übersteigt, da $p(\lambda) := \det(A - \lambda I)$ ein Polynom vom Grad n ist und $p(A) = 0$ gilt. Der folgende wichtige Satz macht eine genauere Aussage über die Dimension eines Krylov-Unterraums.

Satz 2.22. *Sei $A \in \mathbf{R}^{n\times n}, v \in \mathbf{R}^n, \mu := \operatorname{grad}(v)$. Dann gilt*

(i.) $A\mathcal{K}_\mu := \mathcal{K}_\mu(A, Av) \subset \mathcal{K}_\mu$ *und* $\mathcal{K}_\mu = \mathcal{K}_m$ *für alle* $m \geq \mu$,

(ii.) $\dim(\mathcal{K}_m) = m \Longleftrightarrow \mu \geq m$,

(iii.) $\dim(\mathcal{K}_m) = \min(m, \mu)$.

Beweis. Siehe [Saa96], Proposition 6.1 und 6.2 oder [Wer98], Satz 2.1. □

Algorithmen, die die beste Approximation $x_m = x_0 + \mathcal{K}_m$ an die exakte Lösung $A^{-1}b$ ermitteln, liefern bei exakter Arithmetik im Fall $m = n$ und $\dim(\mathcal{K}_n) = n$ also auch die exakte Lösung des Problems. Nach spätestens n Schritten terminiert dann ein solches Verfahren, vorausgesetzt, Rundungsfehler machen dies nicht zunichte.

2.3.2 Das Arnoldi-Orthogonalisierungsverfahren

Für das praktische Rechnen ist es sinvoll, eine Orthonormalbasis von $\mathcal{L}_m$ bzw. $\mathcal{K}_m$ vorliegen zu haben. Als Konsequenz daraus wären die Matrizen V und W aus (2.33) orthogonal, was zu einigen Vereinfachungen führt. Ein bekanntes Verfahren zur Bildung einer solchen Orthonormalbasis im Falle $\mathcal{L}_m = \mathcal{K}_m = \mathcal{K}_m(A, v)$ ist das von Arnoldi, das hier in der numerisch stabileren Variante von Gram-Schmidt vorgestellt wird. Ist $\tilde{H}_m \in \mathbf{R}^{(m+1)\times m}$ eine Matrix mit den Einträgen $h_{ij}, i = 1, \ldots, m+1, j = 1, \ldots, m$, so hat es die Form :

Algorithmus 2.7 Gram-Schmidt-Modifiziertes Arnoldi-Verfahren

Require: $v \neq 0$
1: $v_1 \Leftarrow v/\,||v||_2$
2: **for** $j = 1, \ldots, m$ **do**
3: $\quad w_j \Leftarrow Av_j$
4: $\quad$ **for** $i = 1, \ldots, j$ **do**
5: $\quad\quad h_{ij} \Leftarrow v_i^T w_j$
6: $\quad\quad w_j \Leftarrow w_j - h_{ij} v_i$
7: $\quad$ **end for**
8: $\quad h_{j+1,j} \Leftarrow ||w_j||_2$
9: $\quad$ **if** $h_{j+1,j} = 0$ **then**
10: $\quad\quad$ STOP.
11: $\quad$ **end if**
12: $\quad v_{j+1} \Leftarrow w_j / h_{j+1,j}$
13: **end for**

Satz 2.23. *Es werde angenommen, daß Algorithmus 2.7 nicht vor dem m-ten Schritt abbricht. Dann bilden die Vektoren $v_1, \ldots, v_n$ eine Orthonormalbasis des Krylov-Unterraums $\mathcal{K}_m = \mathcal{K}_m(A, v_1)$.*

Beweis. Siehe [Saa96], Proposition 6.4. □

Außerdem gilt folgender wichtige

Satz 2.24. *Sei $V_m \in \mathbf{R}^{n\times m}$ die aus den Spaltenvektoren $v_1, \ldots, v_n$ gebildete Matrix und $H_m \in \mathbf{R}^{m\times m}$ die (Hessenberg-)Matrix, die aus $\tilde{H}_m$ durch Weglassen der letzten Zeile entsteht. Dann gelten folgende Beziehungen :*

$$AV_m = V_m H_m + w_m e_m^T = V_{m+1} \tilde{H}_m \tag{2.37}$$

$$V_m^T A V_m = H_m \tag{2.38}$$

Beweis. Siehe [Saa96], Proposition 6.5. □

Verwendet man diese Ergebnisse, um den Prototyp des Projektionsverfahren zu spezialisieren, dann ergibt sich für die Matrizen V und W aus (2.33) : $V = W = V_m$, und mit $\beta := ||r_0||_2$ für die rechte Seite von (2.34) :

$$V_m^T r_0 = V_m^T(\beta v_1) = \beta e_1.$$

Damit gilt für die Näherungslösung x_m :

$$x_m = x_0 + V_m y_m \quad \text{mit} \quad y_m = H_m^{-1}(\beta e_1). \tag{2.39}$$

Zusammengefaßt ergibt dies die erste Projektionsmethode für lineare Gleichungssysteme, die *Full Orthogonalization Method (FOM).*

Algorithmus 2.8 FOM

```
1:  r ⇐ b − Ax
2:  β ⇐ ||r||_2
3:  v_1 ⇐ r/β
4:  H_m ⇐ 0
5:  for j = 1, ..., m do
6:     w_j ⇐ Av_j
7:     for i = 1, ..., j do
8:        h_ij ⇐ v_i^T w_j
9:        w_j ⇐ w_j − h_ij v_i
10:    end for
11:    h_{j+1,j} ⇐ ||w_j||_2
12:    if h_{j+1,j} = 0 then
13:       Setze m:= j Gehe zu 13.
14:    end if
15:    v_{j+1} ⇐ w_j/h_{j+1,j}
16: end for
17: Subroutine : Bestimme y = (y_i)_{i=1}^m durch Lösen von H_m y = βe_1.
18: for j = 1, ..., m do
19:    x ⇐ x + y_j v_j
20: end for
```

Das Hauptproblem besteht nun natürlich in der Lösung des linearen Gleichungssystems in Zeile 17. Jedoch wählt man m in der Regel sehr viel kleiner als n, so daß man hier ein direktes Verfahren wie das Gaußsche Eliminationsverfahren oder Givens-Rotationen in Verbindung mit einem Rückwärtseinsetzen verwenden kann.

Es existieren verschiedene Varianten von FOM. Die wichtigsten sind *IOM („incomplete orthogonalization method“)* und *DIOM („direct incomplete orthogonalization method“).* Bei beiden werden die Vektoren v_j nicht gegen alle vorherigen Vektoren orthogonalisiert, sondern nur höchstens gegen alle $k \leq m$ vorigen Vektoren. Konkret startet hierbei die Schleife in Zeile 7 nicht bei 1, sondern bei $\max(1, j-k+1)$. Der Nachteil besteht darin, daß die Vektoren $v_1, \ldots v_m$ nun nicht mehr orthogonal, sondern nur noch *quasi-orthogonal* sind. Ein Vorteil ist jedoch, daß die Matrix H_m nur noch Bandstruktur hat. Zusätzlich wird in DIOM die Lösung x_m in jedem Iterationsschritt durch eine Updateformel unterwegs berechnet, so daß man sich das Lösen des Gleichungssystems in Zeile 17 spart. Näheres zu den beiden Algorithmen erfährt man in [Saa96], Abschnitt 642, Seite 155-158.

2.3.3 Das GMRES-Verfahren

Das *GMRES-Verfahren („generalized minimal residual")*, das ursprünglich in [SS86] zum ersten Mal vorgestellt wurde, verwendet die Räume $\mathcal{K}_m = \mathcal{K}_m(A, v_1)$ und $\mathcal{L}_m = A\mathcal{K}_m$ mit $v_1 = r_0/\left|\left|r_0\right|\right|_2$ für den Projektionsprozeß. Für diesen Fall läßt sich das folgende Optimalitätsresultat angeben :

Satz 2.25. *Seien $A \in \mathbf{R}^{n\times n}$ und $\mathcal{L}_m = A\mathcal{K}_m$. Dann ist $x_m \in \mathbf{R}^n$ genau dann das Ergebnis eines Projektionsverfahrens auf $\mathcal{K}_m$ orthogonal zu $\mathcal{L}_m$ mit der Anfangsnäherung $x_0 \in \mathbf{R}^n$, wenn gilt*

$$\left|\left|b - Ax_m\right|\right|_2 = \min_{x\in x_0+\mathcal{K}_m} \left|\left|b - Ax\right|\right|_2 .$$

Beweis. Siehe z.B. Proposition 5.3 in [Saa96]. □

Da man jeden Vektor x aus $x_0 + \mathcal{K}_m$ als $x_m = x_0 + V_m y$ mit $y \in \mathbf{R}^m$ schreiben kann, ergibt sich unter Verwendung von (2.37) und den Bezeichnungen aus dem Abschnitt über FOM :

$$\begin{aligned} b - Ax &= r_0 - AV_m y \\ &= \beta v_1 - V_{m+1}\tilde{H}_m y \\ &= V_{m+1}(\beta e_1 - \tilde{H}_m y). \end{aligned} \tag{2.40}$$

Berücksichtigt man nun, daß V_{m+1} orthogonal ist, d.h. $V_{m+1}^T V_{m+1} = I$, dann gilt in der Euklidischen Norm

$$\left|\left|b - Ax\right|\right|_2 = \left|\left|b - A(x_0 + V_m y)\right|\right|_2 = \left|\left|\beta e_1 - \tilde{H}_m y\right|\right|_2 . \tag{2.41}$$

GMRES berechnet nun dasjenige $x_m \in x_0 + \mathcal{K}_m$, welches das Residuum aus (2.41) minimiert, was äquivalent dazu ist, ein $y_m \in \mathbf{R}^m$ zu finden, welches $\left|\left|\beta e_1 - \tilde{H}_m y\right|\right|_2$ minimiert.

Algorithmus 2.9 GMRES

1: $r \Leftarrow b - Ax$
2: $\beta \Leftarrow ||r||_2$
3: $v_1 \Leftarrow r/\beta$
4: $\tilde{H}_m \Leftarrow 0$
5: **for** $j = 1, \ldots, m$ **do**
6: $\quad w_j \Leftarrow Av_j$
7: $\quad$ **for** $i = 1, \ldots, j$ **do**
8: $\quad\quad h_{ij} \Leftarrow v_i^T w_j$
9: $\quad\quad w_j \Leftarrow w_j - h_{ij} v_i$
10: $\quad$ **end for**
11: $\quad h_{j+1,j} \Leftarrow ||w_j||_2$
12: $\quad$ **if** $h_{j+1,j} = 0$ **then**
13: $\quad\quad$ Setze m:= j Gehe zu 13.
14: $\quad$ **end if**
15: $\quad v_{j+1} \Leftarrow w_j / h_{j+1,j}$
16: **end for**
17: Subroutine : Transformiere $\tilde{H}_m$ und βe_1 durch Givensrotation Q_m auf $\tilde{R}_m := Q_m \tilde{H}_m$ und $\tilde{g}_m := Q_m \beta e_1$.
18: Subroutine : Bestimme y aus $R_m y = g_m$ durch Rückwärtseinsetzen.
19: **for** $j = 1, \ldots, m$ **do**
20: $\quad x \Leftarrow x + y_j v_j$
21: **end for**

GMRES (vgl. Algorithmus 2.9) unterscheidet sich von FOM also hauptsächlich in Zeile 17, denn hier ist statt dessen ein Least-Squares-Problem zu lösen. Dies ist eindeutig möglich, wenn die Hessenbergmatrix $\tilde{H}_m$ den Rang m hat, was der Fall ist, wenn $h_{j+1,j} \neq 0$ für $j = 1, \ldots, m$ ist. Die Matrix $Q_m \in \mathbf{R}^{(m+1)\times(m+1)}$ ist das Produkt aus den Rotationsmatrizen Ω_i, also

$$Q_m := \Omega_m \Omega_{m-1} \cdots \Omega_1,$$

wobei Ω_i die i-te Givensrotation

$$\Omega_i := \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}$$

ist, mit den Einträgen

$$s_i := \frac{h_{i+1,i}}{\sqrt{(h_{ii}^{(i-1)})^2 + h_{i+1,i}^2}} \quad \text{und} \quad c_i := \frac{h_{ii}^{(i-1)}}{\sqrt{(h_{ii}^{(i-1)})^2 + h_{i+1,i}^2}}.$$

Hierbei ist die Zahl $h_{ii}^{(i-1)}$ der entsprechende Eintrag aus der bereits transformierten Matrix $\Omega_{i-1} \cdots \Omega_1 \tilde{H}_m$. Die Matrix R_m und der Vektor g_m aus Zeile 18 des Algorithmus (2.9) entstehen dann aus $\tilde{R}_m$ bzw. $\tilde{g}_m$ durch Weglassen der jeweils letzten Zeile.

Bemerkung 2.26. *Da die Matrizen Ω_i und damit auch Q_m orthogonal sind, wird die Norm des Residuums nicht verändert, es gilt also, wenn γ_{m+1} die letzte Komponente des Vektors $\tilde{g}_m$ bezeichnet,*

$$\begin{aligned} \left|\left|\beta e_1 - \tilde{H}_m y\right|\right|_2^2 &= \left|\left|Q_m \beta e_1 - Q_m \tilde{H}_m y\right|\right|_2^2 \\ &= \left|\left|\tilde{g}_m - \tilde{R}_m y\right|\right|_2^2 \\ &= |\gamma_{m+1}|^2 + ||g_m - R_m y||_2^2 \,. \end{aligned}$$

Deshalb hat das Residuum der von GMRES erzeugten Lösung die Norm $||b - Ax_m||_2 = \gamma_{m+1}$.

Für den Fall, daß A positiv definit ist, läßt sich eine Abschätzung für das Residuum der gewonnenen Näherungslösung angeben.

Satz 2.27. *Sei $A \in \mathbf{R}^{n \times n}$ positiv definit und $x_m \in \mathbf{R}^n$ die Näherungslösung des GMRES-Verfahrens nach m Schritten mit dem Startvektor x_0. Dann gilt*

$$||b - Ax_m||_2 \leq \left(1 - \frac{\mu^2}{\sigma^2}\right)^{\frac{m}{2}} ||b - Ax_0||_2$$

mit den Konstanten

$$\mu := \frac{1}{2} \min_{i=1,\ldots,n} \left\{ \lambda_i \,|\, \lambda_i \text{ ist Eigenwert von } (A + A^T) \right\} \quad \text{und} \quad \sigma := ||A||_{2,2} \,.$$

Beweis. Sei mit r_0 das Anfangsresiduum $r_0 = b - Ax_0$ bezeichnet. Die Lösung x_m wird als Linearkombination von x_0 und den Vektoren v_i, $i = 1, \ldots, m$ konstruiert, die wiederum eine Darstellung als Linearkombination von Vektoren $r_0, Ar_0, \ldots, A^{m-1} r_0$ besitzen. Deshalb gilt (vgl. [Mei99], Seite 155) mit $r_m = b - Ax_m$:

$$||r_m||_2 \leq \min_{p \in \mathbf{P}_m^1} ||p(A) r_0||_2 \,,$$

wobei $\mathbf{P}_m^1$ die Menge aller Polynome p vom Grad $\leq m$ ist, welche die Nebenbedingung $p(0) = I$ erfüllen. Damit ergibt sich

$$||r_m||_2 \leq \min_{p \in \mathbf{P}_m^1} ||p(A)||_{2,2} \, ||r_0||_2$$

und insbesondere für $p_1(A) := I - \alpha A \in \mathbf{P}_1^1$ mit $\alpha \in \mathbf{R}$ ($\Rightarrow (p_1(A))^m \in \mathbf{P}_m^1$) gilt dann

$$\min_{p \in \mathbf{P}_m^1} ||p(A)||_{2,2} \leq ||(p_1(A))^m||_{2,2} \leq ||p_1(A)||_{2,2}^m \,.$$

Für die Norm von $p_1(A)$ gilt nun nach Definition :

$$||p_1(A)||_{2,2} = \sup_{||x||_2 = 1} ||(I - \alpha A)x||_2 = \sup_{||x||_2 \neq 0} \frac{||(I - \alpha A)x||_2}{||x||_2} .$$

Weiterhin ist

$$\frac{||(I-\alpha A)x||_2^2}{||x||_2^2} = 1 - 2\alpha\frac{x^TAx}{x^Tx} + \alpha^2\frac{x^TA^TAx}{x^Tx},$$

und mit

$$\frac{x^TAx}{x^Tx} = \frac{1}{2}\frac{x^T(A+A^T)x}{x^Tx} \geq \mu > 0$$

und

$$\frac{x^TA^TAx}{x^Tx} \leq \sigma^2$$

erhält man

$$\frac{||(I-\alpha A)x||_2^2}{||x||_2^2} \leq 1 - 2\alpha\mu + \alpha^2\sigma^2. \tag{2.42}$$

Die rechte Seite von (2.42) wird durch $\alpha = \frac{\mu}{\sigma^2}$ minimiert, insbesondere mit diesem α ergibt sich also

$$||p_1(A)||_{2,2} \leq \left(1 - \frac{\mu^2}{\sigma^2}\right)^{\frac{1}{2}}$$

und damit die Behauptung. □

Es gibt auch von GMRES diverse Varianten, die auf einem unvollständigen Orthogonalisierungsprozess beruhen. Verwendet man IOM anstatt der vollen Orthogonalisation, so ergibt sich ein Algorithmus namens *Quasi-GMRES*, verwendet man DIOM, so ergibt sich das sogenannte *DQGMRES*-Verfahren. Beide werden in [SW93a] ausführlich dargestellt. Dort wird deren Konvergenzverhalten außerdem in numerischen Experimenten mit anderen Verfahren wie GMRES verglichen.

Eine wichtige Variante, auf die hier noch näher eingegangen werden soll, ist die Version von GMRES mit Restart, da sie auch in der Klassenbibliothek, die in Kapitel 4 näher behandelt wird, implementiert ist. Sie wird meist als GMRES(m) bezeichnet.

Algorithmus 2.10 GMRES(m)

Require: $\delta > 0$
1: $r \Leftarrow b - Ax$
2: $\beta \Leftarrow ||r||_2$
3: **while** $\beta > \delta$ **do**
4: $v_1 \Leftarrow r/\beta$
5: $\tilde{H}_m \Leftarrow 0$
6: Berechne x wie in Zeile 5 bis 21 von Algorithmus 2.9
7: $r \Leftarrow b - Ax$
8: $\beta \Leftarrow ||r||_2$
9: **end while**

2.3.4 Das Verfahren der konjugierten Gradienten

Das Verfahren der *konjugierten Gradienten*, oder auch *CG-Verfahren („conjugate gradient“)*, ist ein Projektionsverfahren speziell für Gleichungssysteme mit symmetrisch positiv

definiter Koeffizientenmatrix $A \in \mathbf{R}^{n\times n}$. Es läßt sich ebenfalls aus FOM ableiten, wobei sich durch die Symmetrie von A einige Vereinfachungen ergeben. Zunächst läßt sich feststellen, daß die Hessenbergmatrix H_m in diesem Fall symmetrisch ist.

Satz 2.28. *Wird das Orthogonalisierungsverfahren von Arnoldi auf eine symmetrische Matrix $A \in \mathbf{R}^{n\times n}$ angewendet, so gilt*

$$h_{ij} = 0, \quad \textit{für} \quad 1 < i+1 < j \leq m,$$
$$h_{j,j+1} = h_{j+1,j}, \quad \textit{für} \quad j = 1, \ldots, m-1,$$

was gleichbedeutend damit ist, daß H_m eine symmetrische Tridiagonalmatrix ist.

Beweis. Da A symmetrisch ist, gilt mit (2.38) :

$$H_m = V_m^T A V_m = V_m^T A^T V_m = H_m^T,$$

also ist auch H_m symmetrisch. Nach Konstruktion ist H_m aber auch eine Hessenbergmatrix, es ist also für $1 < j+1 < i \leq m$: $h_{ij} = 0 = h_{ji}$, damit verschwinden also auch alle Elemente oberhalb der oberen Nebendiagonalen, und die Aussage ist bewiesen. □

Dies führt, mit den Bezeichnungen $\delta_j := h_{jj}$ und $\eta_j := h_{j-1,j}$, auf eine Variante des Arnoldi-Verfahrens (Algorithmus 2.7), den Lanczos-Algorithmus.

Algorithmus 2.11 Lanczos-Algorithmus

Require: $v \neq 0$
1: $v_1 \Leftarrow v/\,||v||_2$
2: $\eta_1 \Leftarrow 0$
3: $v_0 \Leftarrow 0$
4: **for** $j = 1, \ldots, m$ **do**
5: $\quad w_j \Leftarrow A v_j - \eta_j v_{j-1}$
6: $\quad \delta_j \Leftarrow w_j^T v_j$
7: $\quad w_j \Leftarrow w_j - \delta_j v_j$
8: $\quad \eta_{j+1} \Leftarrow ||w_j||_2$
9: $\quad$ **if** $\eta_{j+1} = 0$ **then**
10: $\quad\quad$ STOP.
11: $\quad$ **end if**
12: $\quad v_{j+1} \Leftarrow w_j/\eta_{j+1}$
13: **end for**

Baut man diesen Algorithmus sinngemäß in FOM ein, ergibt sich das *Lanczos-Verfahren für lineare Gleichungssysteme*, auf das hier jedoch nicht näher eingegangen wird. Es wird lediglich festgehalten, daß das Residuum der von diesem Verfahren erzeugten Lösung x_m dieselbe Richtung wie der Vektor v_{m+1} hat. Auch hier wird nämlich x_m wie in (2.39) berechnet, und deshalb gilt mit $\beta = ||r_0||_2$ und $v_1 = r_0/\beta$:

$$\begin{aligned} b - Ax_m &= b - Ax_0 - AV_m y_m \\ &= r_0 - (V_m H_m + w_m e_m^T) y_m \quad \text{nach (2.37)} \\ &= \beta v_1 - V_m H_m y_m - w_m e_m^T y_m \\ &= \beta v_1 - V_m H_m (H_m^{-1} \beta e_1) - \eta_{m+1} v_{m+1} e_m^T y_m \\ &= -\eta_{m+1} e_m^T y_m v_{m+1}. \end{aligned}$$

Würde man diese Residuen sukzessive berechnen, so wären sie also allesamt orthogonal zueinander. Aus der LU-Zerlegung von H_m lassen sich Update-Formeln für x_m in der Form

$$x_{j+1} := x_j + \alpha_j p_j \tag{2.43}$$

mit einer gewissen Suchrichtung $p_j \in \mathbf{R}^n$ und einer Schrittweite $\alpha_j \in \mathbf{R}$ gewinnen. Dies führt auf das *direkte Lanczos-Verfahren*, auch *D-Lanczos* genannt (siehe [Saa96], Abschnitt 6.7.1). Für die Vektoren p_j läßt sich zeigen, daß sie konjugiert sind, d.h. es gilt $p_i^T A p_j = 0$ für $i \neq j$.

Fordert man nun die Orthogonalität der Vektoren r_j und die Konjugiertheit der Hilfsvektoren p_j und nutzt gleichzeitig noch die positive Definitheit von A aus, dann ergibt sich das Verfahren der konjugierten Gradienten. Ausgehend von (2.43) erhält man

$$r_{j+1} = r_j - \alpha_j A p_j. \tag{2.44}$$

Die Forderung nach der Orthogonalität der Residuen ist also äquivalent zu $(r_j - \alpha_j A p_j)^T r_j = 0$, und daraus folgt

$$\alpha_j = \frac{r_j^T r_j}{r_j^T A p_j}. \tag{2.45}$$

Werden die p_j nun nach der Updateformel

$$p_{j+1} = r_{j+1} + \beta_j p_j \tag{2.46}$$

bestimmt, dann muß gelten :

$$p_{j+1}^T A p_j = (r_{j+1} + \beta_j p_j)^T A p_j = 0 \quad \Longleftrightarrow \quad \beta_j = -\frac{r_{j+1}^T A p_j}{p_j^T A p_j}.$$

Weiterhin ergibt sich durch die Forderung $p_{j-1}^T A p_j = 0$ die Beziehung

$$r_j^T A p_j = (p_j - \beta_{j-1} p_{j-1})^T A p_j = p_j^T A p_j,$$

wodurch man α_j als

$$\alpha_j = \frac{r_j^T r_j}{p_j^T A p_j} \tag{2.47}$$

schreiben kann. Schließlich läßt sich durch (2.44) der Term $A p_j$ in der Gleichung für β_j eliminieren :

$$\beta_j = \frac{r_{j+1}^T r_{j+1}}{r_j^T r_j}. \tag{2.48}$$

Die Formeln (2.43) bis (2.48) ergeben nun den folgenden Algorithmus :

Algorithmus 2.12 CG

Require: $\delta > 0$

1: $r \Leftarrow b$
2: $r \Leftarrow r - Ax$
3: $p \Leftarrow r$
4: $r_S \Leftarrow r^T r$
5: **while** $\sqrt{r_S} > \delta$ **do**
6: $\quad p_A \Leftarrow Ap$
7: $\quad \alpha \Leftarrow r_S / p_A^T p$
8: $\quad x \Leftarrow x + \alpha p$
9: $\quad r \Leftarrow r - \alpha p_A$
10: $\quad r_{S,old} \Leftarrow r_S$
11: $\quad r_S \Leftarrow r^T r$
12: $\quad \beta \Leftarrow r_S / r_{S,old}$
13: $\quad p \Leftarrow r + \beta p$
14: **end while**

Beim CG-Verfahren benötigt man außer für die Problemdaten lediglich Speicherplatz für die Vektoren r, p, p_A und x, denn es wird auf das explizite Bilden einer Orthonormalbasis verzichtet. Statt dessen wird die Lösung x_m als Linearkombination $x_m = x_0 + \sum_{j=0}^{m-1} \lambda_j r_j$ mit gewissen $\lambda_j \in \mathbf{R}$ konstruiert. Gleichzeitig ist aber auch $r_m^T r_j = 0, j = 1, \ldots, m-1$. Das Verfahren der konjugierten Gradienten ist also ein Projektionsverfahren mit $\mathcal{K}_m = \mathcal{L}_m = \mathcal{K}_m(A, r_0)$ und $b - Ax_m \perp \mathcal{L}_m$, deshalb kann man den folgenden Satz darauf anwenden.

Satz 2.29. *Seien $A \in \mathbf{R}^{n \times n}$ symmetrisch positiv definit und $\mathcal{L}_m = \mathcal{K}_m$. Dann ist $x_m \in \mathbf{R}^n$ genau dann das Ergebnis eines orthogonalen Projektionsverfahrens auf $\mathcal{K}_m$ mit der Anfangsnäherung $x_0 \in \mathbf{R}^n$, wenn gilt*

$$||x_* - x_m||_A = \min_{x \in x_0 + \mathcal{K}_m} ||x_* - x||_A ,$$

wobei $x_* = A^{-1}b$ die exakte Lösung bezeichnet.

Beweis. Ein Vektor $x_m \in x_0 + \mathcal{K}_m$ ist genau dann das Ergebnis einer Projektion, wenn die Galerkin-Bedingungen gelten :

$$(b - Ax_m)^T v = 0 \quad \forall v \in \mathcal{L}_m.$$

Andererseits ist es ein Resultat der Approximationstheorie, daß für einen linearen Teilraum $\mathcal{K}$ des Hilbertraumes $\mathbf{R}^n$ (mit dem Skalarprodukt $\langle x, y \rangle_A := x^T Ay$) und ein Element $\tilde{x} \in \mathcal{K}$ gilt : $\tilde{x}$ ist genau dann die beste Approximation eines Elementes $x_* \in \mathbf{R}^n$, wenn $\langle x_* - \tilde{x}, v \rangle_A = 0$ für alle $v \in \mathcal{K}$ gilt (siehe z.B. [SW93b], Satz 14.2.4). Für das x_m aus der Projektion gilt aber gerade dies, denn :

$$\langle x_* - x_m, v \rangle_A = (A(x_* - x_m))^T v = (b - Ax_m)^T v = 0 \quad \forall v \in \mathcal{K}_m = \mathcal{L}_m.$$

□

Das CG-Verfahren minimiert also in jedem Schritt den Fehler $||x_* - x||_A$ für $x \in x_0 + \mathcal{K}_m(A, r_0)$, so daß bei exakter Arithmetik nach n Schritten gilt $x_n = x_*$. Man erhofft sich aber schon für kleine Iterationszahlen eine gute Näherung. Eine Aussage über deren Güte macht der nachfolgende Satz.

Satz 2.30. *Ist x_m die Näherungslösung aus dem m-ten Schritt des CG-Verfahrens, x_0 die Startnäherung und x_* die exakte Lösung, dann gilt*

$$||x_* - x_m||_A \leq 2 \left(\frac{\sqrt{\kappa_2} - 1}{\sqrt{\kappa_2} + 1} \right)^m ||x_* - x_0||_A .$$

Hierbei ist $\kappa_2 = \frac{\lambda_{max}}{\lambda_{min}}$ die Kondition von A bezüglich der Spektralnorm, λ_{max} der größte, λ_{min} der kleinste Eigenwert von A.

Beweis. Siehe z.B. [Saa96], Theorem 6.6. □

2.4 Vorkonditionierung

Wie man unter anderem in Satz 2.30 gesehen hat, ist die Kondition einer Matrix von entscheidender Bedeutung für die Konvergenzgeschwindigkeit eines Iterationsverfahrens. Ist $\kappa(A) \gg 1$ bzw. $\lambda_{max}(A) \gg \lambda_{min}(A)$, so spricht man von einem schlecht konditionierten System. Würde es sich bei A um die Einheitsmatrix handeln, so würde ein Verfahren dagegen sehr schnell konvergieren. Man versucht deshalb, durch geeignete *Vorkonditionierung* die Kondition des linearen Gleichungssystems $Ax = b$ zu verbessern, genauer gesagt, es in ein äquivalentes, aber besser konditioniertes zu transformieren.

Man unterscheidet hierbei drei Strategien :

- **Links-Vorkonditionierung**
  Hierbei wird die Ausgangsgleichung von links mit einer Matrix $P_L \in \mathbf{R}^{n \times n}$ multipliziert, so daß sich das System
  $$P_L A x = P_L b \tag{2.49}$$
  ergibt. Man fordert, daß P_L invertierbar ist und daß $P_L A \approx I$.

- **Rechts-Vorkonditionierung**
  Durch die Transformation $x = P_R y$, mit einer regulären Matrix $P_R \in \mathbf{R}^{n \times n}$ und $u \in \mathbf{R}^n$, ergibt sich das rechts-vorkonditionierte System
  $$A P_R u = b. \tag{2.50}$$
  Auch hier erwartet man, daß $A P_R \approx I$.

- **Beidseitige Vorkonditionierung**
  Die simultane Anwendung einer linken und und einer rechten Vorkonditionierungsmatrix, $P_L \in \mathbf{R}^{n \times n}$ und $P_R \in \mathbf{R}^{n \times n}$, ergibt das System
  $$P_L A P_R u = P_L b \tag{2.51}$$
  mit $u \in \mathbf{R}^n$. Hier wird man fordern, daß $P_L A P_R \approx I$ gilt.

Mit den Vorkonditionierungsmatrizen versucht man also eine Art Approximation der Inversen von A zu erreichen. Meist liegen die Matrizen PL und P_R nicht direkt vor, sondern lediglich ihre Inversen, also eine Näherung an A. Jedoch sollten sie die Forderung nach einer einfachen Invertierbarkeit erfüllen, was zum Beispiel auf Dreiecks- und Diagonalmatrizen zutrifft.

In diesem Abschnitt wird zunächst auf die Vorkonditionierung der CG- und GMRES-Verfahren eingegangen, bevor konkrete Vorkonditionierer beschrieben werden.

2.4.1 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 gewährleisten, ist eine beidseitige Vorkonditionierung mit $P_L = P_R^T$. Man kann jedoch auch eine Variante des CG-Verfahrens entwickeln, die mit Links- oder Rechts-Vorkonditionierung auskommt.

Lemma 2.31. *Sind $P_L, P_R \in \mathbf{R}^{n\times n}$ symmetrisch positiv definite Matrizen, dann gilt :*

(i.) Die Matrix $P_L A$ ist symmetrisch (bzw. selbst-adjungiert) und positiv definit bezüglich des durch

$$\langle x, y\rangle_{P_L^{-1}} := (P_L^{-1}x)^T y \tag{2.52}$$

definierten Skalarproduktes $\langle\cdot,\cdot\rangle_{P_L^{-1}}$.

(ii.) Die Matrix AP_R ist symmetrisch (bzw. selbst-adjungiert) und positiv definit bezüglich des durch

$$\langle x, y\rangle_{P_R} := (P_R x)^T y \tag{2.53}$$

definierten Skalarproduktes $\langle\cdot,\cdot\rangle_{P_R}$.

Beweis. Sei $x, y \in \mathbf{R}^n$. Zu (i.) : Es gilt

$$\langle P_L Ax, y\rangle_{P_L^{-1}} = (Ax)^T y = x^T Ay = x^T P_L^{-1} P_L Ay = (P_L^{-1}x)^T P_L Ay = \langle x, P_L Ay\rangle_{P_L^{-1}},$$

also ist $P_L A$ selbst-adjungiert. Aus $\langle P_L Ax, x\rangle_{P_L^{-1}} = x^T Ax > 0$ folgt die positive Definitheit.
Zu (ii.) : In ähnlicher Weise folgt aus

$$\langle AP_R x, y\rangle_{P_R} = (P_R AP_R x)^T y = (P_R x)^T AP_R y = \langle x, AP_R y\rangle_{P_R},$$

daß die Matrix AP_R selbst-adjungiert und positiv definit ist, da die Matrix $P_R AP_R$ diese Eigenschaften ebenfalls besitzt. □

Benutzt man nun das energetische Skalarprodukt $\langle\cdot,\cdot\rangle_{P_L^{-1}}$ aus (2.52) anstelle des euklidischen zur Beschreibung der Orthogonalitätsbedingungen des CG-Verfahrens für die Matrix $P_L A$ und das transformierte Residuum $z_j := P_L r_j$, dann ergibt sich das (links-) vorkonditionierte CG-Verfahren (*PCG, „preconditioned conjugate gradient“*).

Algorithmus 2.13 PCG

Require: $\delta > 0$

1: $r \Leftarrow b$
2: $r \Leftarrow r - Ax$
3: $z \Leftarrow P_L r$
4: $p \Leftarrow z$
5: $r_S \Leftarrow r^T z$
6: **while** $||r|| > \delta$ **do**
7: $\quad p_A \Leftarrow Ap$
8: $\quad \alpha \Leftarrow r_S / p_A^T p$
9: $\quad x \Leftarrow x + \alpha p$
10: $\quad r \Leftarrow r - \alpha p_A$
11: $\quad z \Leftarrow P_L r$
12: $\quad r_{S,old} \Leftarrow r_S$
13: $\quad r_S \Leftarrow r^T z$
14: $\quad \beta \Leftarrow r_S / r_{S,old}$
15: $\quad p \Leftarrow z + \beta p$
16: **end while**

Verwendet man nun das in (2.53) definierte Skalarprodukt sinngemäß wie zuvor für die Matrix AP_R, so führt dies auf denselben Algorithmus (2.13), lediglich mit P_R in der Rolle von P_L.

Bemerkung 2.32. *Bei PCG hat man, anders als beim normalen CG-Verfahren, keine Kontrolle über die Euklidische Norm des Residuums. Statt dessen wird in jedem Schritt als Nebenprodukt der Wert $r_S = ||r||_A^2$ berechnet, so daß es sich anbietet, die Energie-Norm des Residuums zu überprüfen. Andernfalls muß man die Euklidische Norm von r mit entsprechendem Aufwand von $O(n)$ Operationen gesondert berechnen.*

2.4.2 Das vorkonditionierte GMRES-Verfahren

Betrachtet man für das GMRES-Verfahren zunächst die Vorkonditionierung von links, dann wird die Orthonormalbasis des Krylov-Unterraums anstatt mit $r_0 = b - Ax_0$ nun mit dem transformierten Residuum $z_0 := P_L r_0$ gebildet. Ersetzt man des weiteren in Algorithmus 2.9 die Matrix A durch $P_L A$, dann ergibt sich das in Algorithmus 2.14 dargelegte Verfahren .

Auch hier hat man keinen direkten Zugriff auf das nicht vorkonditionierte Residuum, um es für eine Abbruchkontrolle zu benutzen. Jedoch wird man auch hier, wie in den meisten Fällen, mit Restart arbeiten, so daß man das neue Residuum ohnehin berechnen muß, um einen weiteren GMRES-Durchlauf zu starten.

Algorithmus 2.14 GMRES mit Links-Vorkonditionierung

1: $z \Leftarrow b - Ax$
2: $z \Leftarrow P_L z$
3: $\beta \Leftarrow ||z||_2$
4: $v_1 \Leftarrow z/\beta$
5: $\tilde{H}_m \Leftarrow 0$
6: **for** $j = 1, \ldots, m$ **do**
7: $\quad w_j \Leftarrow Av_j$
8: $\quad w_j \Leftarrow P_L w_j$
9: $\quad$ **for** $i = 1, \ldots, j$ **do**
10: $\quad\quad h_{ij} \Leftarrow v_i^T w_j$
11: $\quad\quad w_j \Leftarrow w_j - h_{ij} v_i$
12: $\quad$ **end for**
13: $\quad h_{j+1,j} \Leftarrow ||w_j||_2$
14: $\quad$ **if** $h_{j+1,j} = 0$ **then**
15: $\quad\quad$ Setze m:= j Gehe zu 13.
16: $\quad$ **end if**
17: $\quad v_{j+1} \Leftarrow w_j / h_{j+1,j}$
18: **end for**
19: Subroutine : Transformiere $\tilde{H}_m$ und βe_1 durch Givensrotation Q_m auf $\tilde{R}_m := Q_m \tilde{H}_m$ und $\tilde{g}_m := Q_m \beta e_1$.
20: Subroutine : Bestimme y aus $R_m y = g_m$ durch Rückwärtseinsetzen.
21: **for** $j = 1, \ldots, m$ **do**
22: $\quad x \Leftarrow x + y_j v_j$
23: **end for**

Wendet man eine rechte Vorkonditionierung auf GMRES an, so wird, anders als (2.50) vermuten läßt, die dort eingeführte zusätzliche Variable u im Algorithmus nicht explizit benötigt. Da $P_R u_0 = x_0$, ist $r_0 = b - AP_R u_0 = b - Ax_0$. Außerdem ist die GMRES-Approximation zu (2.50) durch

$$u_m = u_0 + \sum_{i=1}^{m} v_i y_i$$

gegeben, woraus man durch Multiplizieren mit P_R die Lösung des ursprünglichen Problems

$$x_m = x_0 + P_R \sum_{i=1}^{m} v_i y_i$$

erhält. Ersetzt man nun in Algorithmus 2.9 alle Matrixmultiplikationen mit A durch Multiplikationen mit AP_R, so ergibt sich die entsprechende rechts vorkonditionierte Variante von GMRES.

Handelt es sich bei P_L um eine symmetrisch positiv definite (linke) Vorkonditionierungsmatrix, so läßt sich eine Variante von GMRES entwickeln, die von diesen Eigenschaften Gebrauch macht (siehe z.B. [Saa96], Seite 252-253). Für dieses Verfahren, welches

das Residuum $||P_L r_m||_{P_L^{-1}}$ minimiert, läßt sich eine mit Satz 2.27 vergleichbare Aussage formulieren.

Satz 2.33. *Sei $P_L \in \mathbf{R}^{n\times n}$ eine symmetrisch positiv definite linke Vorkonditionierungsmatrix. Mit den Bezeichnungen*

$$\tau = \inf_{\substack{w\in\mathbf{R}^n \\ w\neq 0}} \frac{w^T A w}{w^T P_L^{-1} w}$$

$$\tilde{\tau} = \inf_{\substack{w\in\mathbf{R}^n \\ w\neq 0}} \frac{w^T A^{-1} w}{w^T P_L w}$$

gilt die Abschätzung

$$||P_L r_m||_{P_L^{-1}} \leq (1-\tau\tilde{\tau})^{\frac{m}{2}} \, ||P_L r_0||_{P_L^{-1}} .$$

Beweis. Siehe [Sta97], Seite 109, Theorem 3.2. □

2.4.3 Splitting-Methoden als Vorkonditionierer

In Abschnitt 2.2 wurden diverse Splitting-Methoden und ihre Eigenschaften vorgestellt. Da sie aber im Vergleich zu vielen Krylov-Methoden schlechtere Konvergenzeigenschaften aufweisen, werden sie als reine Lösungsverfahren immer seltener eingesetzt. Von großer Bedeutung sind sie jedoch, wenn man sie in Verbindung mit den Projektions-Verfahren als Vorkonditionierer anwendet. Denn da sie auf einer Zerlegung $A = M - N$ basieren, bei der M eine leicht invertierbare Approximation von A darstellt, liegt es nahe, die Matrix M^{-1} als Vorkonditionierungsmatrix zu verwenden.

Definition 2.34 (Splitting-assoziierte Vorkonditionierer). *Ist durch die Iterationsvorschrift $x^{(k+1)} := M^{-1}Nx^{(k)} + M^{-1}b$ eine Splitting-Methode zur Lösung von $Ax = b$ gegeben, so heißt $P = M^{-1}$ die zur Splitting-Methode assoziierte Vorkonditionierungsmatrix.*

Mit der durch (2.15) eingeführten Zerlegung ergibt sich die in Tabelle 2.1 dargestellte Übersicht über diese Klasse von Vorkonditionierern.

Splitting-Methode	**Assoziierte Vorkonditionierungsmatrix**
Jacobi-Verfahren	$P_J = D^{-1}$
Gauß-Seidel-Verfahren	$P_{GS} = (L+D)^{-1}$
Symmetrisches Gauß-Seidel-Verfahren	$P_{SGS} = (D+R)^{-1}D(L+D)^{-1}$
JOR-Verfahren	$P_{JOR}(\omega) = \omega D^{-1}$
SOR-Verfahren	$P_{SOR}(\omega) = \omega(\omega L + D)^{-1}$
SSOR-Verfahren	$P_{SSOR}(\omega) = \omega(2-\omega)(D+\omega R)^{-1}D(\omega L + D)^{-1}$

Tabelle 2.1: Zu Splitting-Methoden assoziierte Vorkonditionierungsmatrizen

Der Jacobi-Vorkonditionierer z.B. ist also nichts anderes als die Multiplikation mit dem Inversen der Diagonalen von A. Da die inversen Diagonalelemente bei jedem dieser Verfahren benötigt werden, sind sie in der Klassenbibliothek so implementiert, daß tatsächlich

eine Diagonalmatrix D^{-1} angelegt wird, d.h. der Jacobi-Vorkonditionierer ist der einzige, der explizit gegeben ist. Alle anderen sind implizit, also durch ihre *Wirkung* bei Matrixmultiplikationen, gegeben. Untere Dreiecksmatrizen werden durch Vorwärtseinsetzen, obere durch Rückwärtseinsetzen invertiert. Bei Produkten aus mehreren solchen Matrizen (wie beispielsweise P_{SGS}) wird dies durch die Nacheinanderausführung dieser Techniken erreicht.
Im Falle einer symmetrischen Matrix A sind die zum Jacobi-, zum Symmetrischen Gauß-Seidel- und zum SSOR-Verfahren assoziierten Vorkonditionierungsmatrizen offensichtlich selbst wieder symmetrisch, so daß sie sich zur Vorkonditionierung des CG-Verfahrens eignen.

2.4.4 Die Unvollständige LU-Zerlegung

Die Faktorisierung einer Matrix $A \in \mathbf{R}^{n\times n}$ in eine linke untere Dreiecksmatrix $L \in \mathbf{R}^{n\times n}$ und eine rechte obere Dreiecksmatrix $U \in \mathbf{R}^{n\times n}$ mit

$$A = LU, \tag{2.54}$$

wie sie z.B. in [Mei99], Abschnitt 3.1 oder [Lub98a], Abschnitt 13.2 beschrieben wird, ist eine direkte Methode und als Lösungsverfahren nur für Matrizen mit spezieller Gestalt (z.B. Bandstruktur) sinnvoll. Denn L und U sind in der Regel nicht dünnbesetzt, selbst wenn A diese Eigenschaft hat, so daß die Durchführung einer solchen Zerlegung für allgemeine, große dünnbesetzte Matrizen A aus Speicherplatz- und Rechenzeitgründen keinen Sinn hat, da hinter der LU-Zerlegung ja das Gaußsche Eliminationsverfahren steckt. Es ergibt sich jedoch ein guter Vorkonditionierer, wenn man L und U nur näherungsweise berechnet. Hierbei wird (2.54) zu $A = LU + F$ abgeschwächt, wobei $F \in \mathbf{R}^{n\times n}$ die Rest- oder Fehlermatrix ist. Außerdem sollen L und U zusammen nur soviel Speicherplatz wie A verbrauchen. Hierbei sind die folgenden Begriffe hilfreich.

Definition 2.35 (Besetzungsmuster). *Die Menge*

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

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

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

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

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

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

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

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

Mit Hilfe des Besetzungsmusters läßt sich nun die unvollständige LU-Zerlegung (*incomplete LU decomposition, ILU*) definieren.

Definition 2.36 (ILU-Zerlegung). *Sei* $A = (a_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ *eine reguläre Matrix,* $L = (l_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ *eine reguläre linke untere Dreiecksmatrix und* $U = (u_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ *eine reguläre rechte obere Dreiecksmatrix. Die Zerlegung*

$$A = LU + F \tag{2.55}$$

existiere unter den Bedingungen

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

Dann heißt (2.55) unvollständige LU-Zerlegung *der Matrix A zum Muster* $\mathcal{M}^A$.

Aus dieser Definition ergeben sich die Formeln

$$l_{ki} = a_{ki} - \sum_{\substack{m=1 \\ m\in\mathcal{M}_{\mathcal{Z}}^A(k)\cap\mathcal{M}_{\mathcal{S}}^A(i)}}^{i-1} l_{km}u_{mi} \quad \text{für} \quad k = i,\ldots,n, \quad k \in \mathcal{M}_{\mathcal{S}}^A(i) \tag{2.56}$$

für die i-te Spalte von L und

$$u_{ik} = \frac{1}{l_{ii}}\left(a_{ik} - \sum_{\substack{m=1 \\ m\in\mathcal{M}_{\mathcal{Z}}^A(i)\cap\mathcal{M}_{\mathcal{S}}^A(k)}}^{i-1} l_{im}u_{mk}\right) \quad \text{für} \quad k = i+1,\ldots,n, \quad k \in \mathcal{M}_{\mathcal{Z}}^A(i) \tag{2.57}$$

für die i-te Zeile von U.
Insgesamt gewinnt man aus (2.56) und (2.57) den (vorläufigen) Algorithmus 2.15.

Bemerkung 2.37. *Algorithmus 2.15 ist insofern vorläufig, als er sich noch nicht die spezielle Abspeicherung der in der Klassenbibliothek verwendeten Datenstruktur zunutze macht. Eine hieran angepaßte Version des Algorithmus wird im Unterabschnitt 4.6.2 vorgestellt.*

Algorithmus 2.15 ILU

```
1: for i = 1, ..., n do
2:   for k = i, ..., n do
3:     if k ∈ M_S^A(i) then
4:       l_ki ⇐ a_ki
5:       for m = 1, ..., i − 1 do
6:         if m ∈ M_Z^A(k) ∩ M_S^A(i) then
7:           l_ki ⇐ l_ki − l_km u_mi
8:         end if
9:       end for
10:    end if
11:  end for
12:  for k = i + 1, ..., n do
13:    if k ∈ M_Z^A(i) then
14:      u_ik ⇐ a_ik
15:      for m = 1, ..., i − 1 do
16:        if m ∈ M_Z^A(i) ∩ M_S^A(k) then
17:          u_ik ⇐ u_ik − l_im u_mk
18:        end if
19:      end for
20:      u_ik ⇐ u_ik / l_ii
21:    end if
22:  end for
23: end for
```

Definition 2.38 (ILU-Vorkonditionierer). *Ist die ILU-Zerlegung der Matrix A wie in (2.55) gegeben, so ist*

$$P := U^{-1}L^{-1} \tag{2.58}$$

die zugehörige Vorkonditionierungsmatrix.

Die Invertierung der Dreiecksmatrizen L und U geschieht in der Praxis wie üblich durch Vorwärts- bzw. Rückwärtseinsetzen.

Die Frage nach der Existenz einer solchen ILU-Zerlegung läßt sich zumindest für *M-Matrizen* eindeutig beantworten.

Definition 2.39 (M-Matrix). *Sei $A = (a_{ij})_{i,j=1}^n \in \mathbf{R}^{n\times n}$ eine reguläre Matrix mit $A^{-1} = (\tilde{a}_{ij})_{i,j=1}^n$. Man nennt A eine* M-Matrix, *wenn die folgenden Bedingungen erfüllt sind :*

(i.) $a_{ii} > 0$ *für* $i = 1, \ldots, n$,

(ii.) $a_{ij} \leq 0$ *für* $i, j = 1, \ldots, n$, $i \neq j$,

(iii.) $\tilde{a}_{ij} \geq 0$ *für* $i, j = 1, \ldots, n$.

Zu einer Matrix $A \in \mathbf{R}^{n\times n}$ und einem Muster $\mathcal{M}$ im $\mathbf{R}^{n\times n}$ bezeichnet man nun die Menge

$$\mathcal{P}_{\mathcal{M}}^{A} := \{(i,j) \in \mathcal{M} \,|\, (i,j) \notin \mathcal{M}^{A}, i \neq j\}$$

als Nullmuster.

Satz 2.40 (Meijerink und van der Vorst). *Sei $A \in \mathbf{R}^{n\times n}$ eine M-Matrix und $\mathcal{P}_{\mathcal{M}}^{A}$ ein Nullmuster. Dann ist die ILU-Zerlegung durchführbar, und es existieren genau eine reguläre linke untere Dreiecksmatrix L und eine reguläre rechte obere Dreiecksmatrix U mit den Eigenschaften $A = LU - F$ und*

$$l_{ij} = u_{ij} = 0 \quad \textit{für } (i,j) \in \mathcal{P}_{\mathcal{M}}^{A} \qquad \textit{und} \qquad r_{ij} = 0 \quad \textit{für } (i,j) \notin \mathcal{P}_{\mathcal{M}}^{A},$$

wobei $F = (f_{ij})_{i,j=1}^{n} \in \mathbf{R}^{n\times n}$ die Fehlermatrix ist.

Beweis. Siehe [Gre97], Seite 173, Theorem 11.1.1 oder [Mv77], Seite 150, Theorem 2.3. □

Bemerkung 2.41. *In [Gre97] und [Mv77] wird eigentlich sogar eine weitergehende Aussage bewiesen, nämlich die Existenz einer ILU-Zerlegung für jedes Nullmuster $\mathcal{P} \subset \mathcal{P}_{\mathcal{M}}^{A}$. Ist $\mathcal{P}$ eine echte Teilmenge von $\mathcal{P}_{\mathcal{M}}^{A}$, so hat man weniger von Null verschiedene Einträge in der Fehlermatrix F und somit eine bessere Approximation von $A \approx LU$. Dies führt auf die Klasse der* ILU(p)-Verfahren, *bei denen berücksichtigt wird, daß bei L und U mehr von Null verschiedene Elemente entstehen. Hierbei ist $p \in \mathbf{N}$ ein Grad für die Anzahl dieser zusätzlichen* fill-in-*Elemente.*

2.4.5 Die Unvollständige Cholesky-Zerlegung

Ähnlich wie im vorigen Kapitel läßt sich nun das entsprechende Pendant zur Cholesky-Zerlegung, wie sie beispielsweise in [Mei99], Abschnitt 3.2 dargestellt wird, entwickeln : die *unvollständige Cholesky-Zerlegung* für symmetrisch positiv definite Matrizen.

Definition 2.42 (IC-Zerlegung). *Sei $A \in \mathbf{R}^{n\times n}$ symmetrisch und positiv definit und $L = (l_{ij})_{i,j=1}^{n} \in \mathbf{R}^{n\times n}$ eine reguläre linke untere Dreiecksmatrix. Die Zerlegung*

$$A = LL^{T} + F \tag{2.59}$$

existiere unter den Bedingungen

- $l_{ij} = 0$, *falls* $(i,j) \notin \mathcal{M}^{A}$,
- $(LL^{T})_{ij} = a_{ij}$, *falls* $(i,j) \in \mathcal{M}^{A}$.

Dann heißt (2.59) unvollständige Cholesky-Zerlegung (IC, „incomplete Cholesky“) *der Matrix A zum Muster $\mathcal{M}^{A}$.*

Daraus ergibt sich für die Diagonalelemente

$$l_{kk} = \sqrt{a_{kk} - \sum_{\substack{j=1 \\ j\in\mathcal{M}_{\mathcal{Z}}^{A}(k)}}^{k-1} l_{kj}^{2}} \quad \text{für} \quad k = i,\ldots,n, \tag{2.60}$$

und darauf aufbauend für die i-te Zeile von L

$$l_{ik} = \frac{1}{l_{kk}} \left(a_{ik} - \sum_{\substack{j=1 \\ j \in \mathcal{M}_{\mathcal{Z}}^{A}(i) \cap \mathcal{M}_{\mathcal{Z}}^{A}(k)}}^{k-1} l_{ij} l_{kj} \right) \quad \text{für} \quad i = k+1, \ldots, n, \quad i \in \mathcal{M}_{\mathcal{Z}}^{A}(k), \tag{2.61}$$

womit man den allgemeinen Algorithmus 2.16 formulieren kann.

Algorithmus 2.16 IC

```
1: for k = 1, ..., n do
2:    l_kk ⇐ a_kk
3:    for j = 1, ..., k − 1 do
4:       if j ∈ M_Z^A(k) then
5:          l_kk ⇐ l_kk − l_kj^2
6:       end if
7:    end for
8:    l_kk ⇐ √l_kk
9:    for i = k + 1, ..., n do
10:      if i ∈ M_Z^A(k) then
11:         l_ik ⇐ a_ik
12:         for j = 1, ..., k − 1 do
13:            if j ∈ M_Z^A(i) ∩ M_Z^A(k) then
14:               l_ik ⇐ l_ik − l_ij l_kj
15:            end if
16:         end for
17:         l_ik ⇐ l_ik / l_kk
18:      end if
19:   end for
20: end for
```

Da die Matrix LL^T und das Produkt LAL^T symmetrisch und positiv definit sind, eignet sich die unvollständige Cholesky-Zerlegung sowohl für links- und rechts- als auch für beidseitige Vokonditionierung, bei der es auf die Erhaltung dieser Eigenschaften ankommt, wie beispielsweise beim CG-Verfahren.

Zum Schluß sei auch für die IC-Zerlegung eine Existenzaussage, ähnlich zu Satz 2.40, angegeben.

Satz 2.43. *Sei $A \in \mathbf{R}^{n \times n}$ eine symmetrische M-Matrix und $\mathcal{P}_{\mathcal{M}}^{A}$ ein Nullmuster. Dann ist die IC-Zerlegung durchführbar. Es existiert genau eine reguläre linke untere Dreiecksmatrix L mit den Eigenschaften $A = LL^T - F$ und*

$$l_{ij} = 0 \quad \textit{für } (i,j) \in \mathcal{P}_{\mathcal{M}}^{A} \qquad \textit{und} \qquad r_{ij} = 0 \quad \textit{für } (i,j) \notin \mathcal{P}_{\mathcal{M}}^{A},$$

wobei $F = (f_{ij})_{i,j=1}^{n} \in \mathbf{R}^{n \times n}$ die Fehlermatrix ist.

Beweis. Siehe [Gre97], Seite 173, Korollar 11.1.1 oder [Mv77], Seite 151, Theorem 2.4. □

2.4.6 Vorkonditionierung mit dem symmetrischen Anteil

Die bisher dargestellten Vorkonditionierer nutzen die spezielle Struktur des linearen Gleichungssystems kaum aus, lediglich die Symmetrie wird beim CG-Verfahren und bei der unvollständigen Cholesky-Zerlegung berücksichtigt. Zieht man jedoch in Betracht, daß das Gleichungssystem $Ax = b$ aus der Diskretisierung von $a(u, v) = f(v)$ (siehe (1.26) und (1.27)) herrührt, so kann man sich unter gewissen Umständen besser an das ursprüngliche Problem angepaßte Vorkonditionierer vorstellen, wie z.B. die folgende Idee der *SPD-Vorkonditionierung*, wie sie in [Sta97] und [ML99] untersucht wurde.

Betrachtet man die Bilinearform $a(\cdot, \cdot)$, so kann man sie in einen symmetrischen Anteil $a^{sym}(\cdot, \cdot)$ und einen schiefsymmetrischen Anteil aufsplitten $a^{skew}(\cdot, \cdot)$:

$$a^{sym}(u, v) := \varepsilon(\nabla u, \nabla v)_\Omega + (cu, v)_\Omega, \tag{2.62}$$

$$a^{skew}(u, v) := \frac{1}{2}\left((\vec{b} \cdot \nabla u, v)_\Omega + (\vec{b} \cdot \nabla v, u)_\Omega\right), \tag{2.63}$$

so daß die Variationsgleichung (1.28) die folgende Formulierung erhält :

$$\textit{Finde } u \in X : \quad a(u, v) \equiv a^{sym}(u, v) + a^{skew}(u, v) = f(v) \quad \forall v \in X := W_0^{1,2}(\Omega).$$

Hat man statt der Standard-Galerkin-Diskretisierung eine andere, wie etwa die Streamline-Diffusion-Diskretisierung, vorliegen, so lassen sich der symmetrische und der schiefsymmetrische Anteil folgendermaßen angeben :

$$a_{SD}^{sym}(u, v) := \frac{1}{2}\left(a_{SD}(u, v) + a_{SD}(v, u)\right), \tag{2.64}$$

$$a_{SD}^{skew}(u, v) := \frac{1}{2}\left(a_{SD}(u, v) - a_{SD}(v, u)\right). \tag{2.65}$$

Den symmetrischen Anteil der Matrix A, im folgenden mit A_{sym} bezeichnet, berechnet man nun analog zu (1.32) :

$$A_{sym} = (a_{ij}^{sym})_{i,j=1}^n \in \mathbf{R}^{n \times n} \quad mit\ a_{ij}^{sym} := a^{sym}(\phi_j, \phi_i). \tag{2.66}$$

Aus der Symmetrie von A_{sym} und Satz 1.30 bzw. 1.33 folgt dann mit den Bezeichnungen aus (1.31) :

$$0 < a(v, v) = a^{sym}(v, v) = \langle A_{sym}\vec{v}, \vec{v}\rangle = \langle \vec{v}, A_{sym}\vec{v}\rangle \quad \forall v \in X_n, \quad \vec{v} \in \mathbf{R}^n,$$

und damit die positive Definitheit von A_{sym}.

Die Idee bei der Vorkonditionierung mit dem symmetrischen (positiv definiten) Anteil (*SPD-Vorkonditionierung*) ist nun auf der Hoffnung begründet, durch A_{sym} eine gute Approximation für A zu haben, bzw. durch A_{sym}^{-1} eine gute Näherung an A^{-1}. Dies ist insbesondere für kleine Peclet-Zahlen, d.h. für diffusionsdominierte Probleme, zu erwarten.

Definition 2.44 (SPD-Vorkonditionierer). *Die Matrix A_{sym}^{-1} mit $A_{sym} = \frac{1}{2}(A + A^T)$ aus (2.66) bezeichnet man als SPD-Vorkonditionierungsmatrix. Bei der SPD-Vorkonditionierung betrachtet man also das System*

$$A_{sym}^{-1}Ax = A_{sym}^{-1}b.$$

Da A nach wie vor unsymmetrisch ist, wird man in der Praxis als Ausgangsverfahren einen Löser wie GMRES einsetzen. Der symmetrische Anteil von A läßt sich schnell (in $O(nnz)$) als $A_{sym} = \frac{1}{2}(A + A^T)$ berechnen. Für die (linke) Vorkonditionierung mit A_{sym}^{-1} benötigt man eine Invertierung von A_{sym}, die man in der Regel mit dem CG-Verfahren durchführen wird. Um eine weitere Beschleunigung zu erreichen, ist es natürlich denkbar und oft auch sinnvoll, dieses CG-Verfahren seinerseits wieder vorzukonditionieren (siehe auch Abschnitt 4.3.2 zur Implementation der Vorkonditionierer und Löser).

Kapitel 3

Datenstrukturen zur effizienten Speicherung von Matrizen

Bei den zur Lösung von partiellen Differentialgleichungen eingesetzten Diskretisierungsverfahren entstehen in der Praxis Koeffizientenmatrizen mit einer sehr großen Dimensionszahl n, die in der Regel umso größer wird, je genauer das Problem gelöst werden soll.
Bei diesen Größenordnungen (bei praktisch relevanten Aufgaben sind Zahlen von $n = 10000$ noch eher wenig) versagen traditionelle Speichertechniken : Nehmen wir für Double-Zahlen einmal eine Länge von 64 Bit an, dann würde eine 10000×10000 - Matrix eine Größe von 763 Megabyte im Speicher belegen. Dies würde auch beim heutigen Stand der Technik nicht nur bedeuten, daß Festplatten schnell gefüllt wären, hinzu käme, daß die Matrix kaum in das RAM des Rechners passen würde und deshalb Operationen mit der Matrix durch ständiges Swappen unvertretbar verlangsamt würden.
Aber nicht nur der Speicherbedarf ist problematisch, auch würde man, bei einer Matrixmultiplikation beispielsweise, erhebliche Zeit mit der wiederholten Multiplikation und Addition von Nullen verschwenden. Aus diesem Grunde liegt es nahe, die Daten der Matrix in einer komprimierten Form abzuspeichern. Im folgenden werden einige dieser Speichertechniken vorgestellt und anhand der Beispielmatrix

$$A = \begin{pmatrix} 5 & 0 & 0 & 4 & 0 & 0 & 0 \\ 3 & 8 & 0 & 6 & 0 & 1 & 0 \\ 0 & 0 & 9 & 0 & 2 & 0 & 0 \\ 0 & 5 & 2 & 1 & 0 & 0 & 5 \\ 0 & 0 & 10 & 0 & 7 & 4 & 0 \\ 0 & 0 & 0 & 4 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 12 \end{pmatrix}$$

erläutert. Die Anzahl der von Null verschiedenen Elemente sei mit nnz bezeichnet, hier also $nnz = 19$. DS sei der Speicher, den eine Gleitkommazahl belegt, IS der Speicherplatz für eine Integer-Variable, n die Dimension der Matrix.

3.1 Das Koordinatenformat (Coordinate Storage)

Das *Koordinatenformat* (*Coordinate Storage*) ist das einfachste denkbare Speicherformat für schwach besetzte Matrizen. Hierbei werden alle Nichtnullelemente hintereinander in einem Array mit Gleitkommazahlen abgespeichert. Hinzu kommen zwei Arrays mit ganzen Zahlen, einer für die zugehörigen Spaltenindizes und einer für die Zeilenindizes. Dabei ist es unerheblich, in welcher Reihenfolge die Elemente abgespeichert werden, denn die z.B. sehr wichtige Operation des Matrix-Vektor-Produktes, die bei iterativen Verfahren oft auftaucht, läßt sich auch ohne eine besondere Ordnung performant implementieren, vorausgesetzt, der Vektor liegt in einem Format vor, das einen wahlfreien Zugriff (*random access*) über Indizes erlaubt. Denn gerade dieser wahlfreie Zugriff auf ein bestimmtes Matrixelement ist beim Koordinatenformat nicht sehr effizient, da für die Suche des Elementes im *worst case* die gesamte Datenstruktur durchlaufen werden muß.

WERT	5	4	3	8	6	1	9	2	5	2	1	5	10	7	4	4	3	3	12
SPALTE	1	4	1	2	4	6	3	5	2	3	4	7	3	5	6	4	6	6	7
ZEILE	1	1	2	2	2	2	3	3	4	4	4	4	5	5	5	6	6	7	7

Tabelle 3.1: Darstellung der Matrix A im Koordinatenformat

Bei einer sortierten Speicherung der Matrixelemente, welche zeilenweise oder reihenweise geschehen kann, läßt sich dieser Zugriff jedoch beschleunigen, wenn man sich z.B. noch merkt, bei welchem Index des Arrays ZEILE eine neue Zeile beginnt. Man erkennt hieran, daß dieses Format noch nicht ganz effizient mit dem Speicher umgeht, und dies führt direkt zum nächsten Speicherformat, dem der komprimierten Zeilen.

Speicherbedarf : $(DS + 2 * IS) * nnz$

3.2 Komprimierte Zeilen (Compressed Row Storage)

Mit dem *Compressed Row Storage* (*CRS*) Format kann man im Vergleich zum Koordinatenformat weiteren Speicherplatz einsparen, indem man sich nur merkt, wann eine neue Zeile beginnt, anstatt für jeden Eintrag die Zeilennummer abzuspeichern. Die Einträge können bei dieser Speichertechnik natürlich nicht mehr ganz beliebig im Array WERTE stehen, vielmehr müssen die Einträge einer Zeile in einem Block hintereinander im Array abgelegt sein. Der Vektor SPALTE enthält wie bisher den zugehörigen Spaltenindex. Im Array ZEILENANFANG stehen die Indizes, bei denen in WERTE (bzw. SPALTE) ein neuer Block einer Zeile beginnt.

Bemerkung 3.1. *Beim CRS wird vorausgesetzt, daß in jeder Zeile mindestens ein Eintrag vorhanden ist. Andernfalls wäre dann die Matrix zum einen natürlich singulär, zum anderen wäre es außerdem problematisch, weil sich dann beispielsweise nicht ohne weiteres*

WERT	5	4	3	8	6	1	9	2	5	2	1	5	10	7	4	4	3	3	12

SPALTE	1	4	1	2	4	6	3	5	2	3	4	7	3	5	6	4	6	6	7

ZEILENANFANG	1	3	7	9	13	16	18	20

Tabelle 3.2: Darstellung der Matrix A im Compressed Row Storage Format

identifizieren ließe, ob auf Zeile k die Zeile $k+1$ oder $k+2$ folgt.

Den letzten Eintrag von ZEILENANFANG setzt man für gewöhnlich auf den Wert $nnz + 1$, was auch für manche Algorithmen vorteilhaft ist, denn es gilt dann immer : Länge der Zeile i = ZEILENANFANG[i+1] - ZEILENANFANG[i]. Aus diesen Gründen hat dieser Array die Länge $n + 1$.

Speicherbedarf : $(DS + IS) * nnz + IS * (n + 1)$

Eine Variante dieses Formats, die in der Literatur und in manchen Softwarepaketen auftaucht, ist das *Modified Sparse Row Storage* (*MSR*) Format. Hierbei wird die Diagonale der Matrix separat abgespeichert, nämlich in den ersten n Komponenten des Arrays WERT, und dahinter dann die anderen Einträge der Matrix, zeilenweise. Die Idee dahinter ist, dass es bei vielen iterativen Verfahren (wie z.B. dem SOR) oder beim Lösen von Hilfsproblemen durch Vorwärts- oder Rückwärtseinsetzen hilfreich ist, einen direkten Zugriff auf die Diagonalelemente zu haben.

3.3 Komprimierte Spalten (Compressed Column Storage)

Das *Compressed Column Storage* (*CCS*) Format speichert die Matrix im Gegensatz zum CRS spaltenweise ab. Im Array WERT stehen also die Spalten blockweise hintereinander, es gibt einen Array ZEILE, der zu jedem Eintrag in WERT den entsprechenden Spaltenindex aufnimmt. Ein Array SPALTENANFANG speichert den Index, bei dem in WERTE eine neue Spalte beginnt. Das CCS-Format ist deshalb nichts anderes als das CRS-Format für die Matrix A^T. Dieses Matrixformat wird auch als *Harwell-Boeing Sparse Matrix Format* bezeichnet.

Das Analogon zum MSR ist hierbei das *Modified Sparse Column Storage, (MSC)*, siehe [Saa94], Abschnitt 2.1.1.

Speicherbedarf : wie beim CRS.

WERT	5	3	8	5	9	2	10	4	6	1	4	2	7	1	4	3	3	5	12

ZEILE	1	2	2	4	3	4	5	1	2	4	6	3	5	2	5	6	7	4	7

SPALTENANFANG	1	3	5	8	12	14	18	20

Tabelle 3.3: Darstellung der Matrix A im Compressed Columns Storage Format

3.4 Komprimierte Diagonalen (Compressed Diagonal Storage)

Definition 3.2. *Man sagt, eine Matrix $A = (a_{i,j})$ habe Bandstruktur, wenn es nichtnegative Konstanten p, q gibt, so daß $a_{i,j} = 0$ für $-p > j - i > q$. Die Zahl $p + q$ wird Bandbreite genannt, während die Zahlen p und q als linke untere und rechte obere Bandbreite bezeichnet werden. Dementsprechend wird die Menge $B_k(A) := \{a_{i,j} | j - i = k\}$, $-p \leq k \leq p$, als Band bezeichnet.*

Hat eine Matrix eine solche Bandstruktur, so kann man sich dies bei der Speicherung zunutze machen. In diesem Fall beschafft man sich Speicher für einen (zweidimensionalen) WERT-Array der Größe $n*(p+q+1)$. Hierin wird die Matrix nun Band für Band abgelegt, und zwar derart, daß

$$WERT(k, i) = a_{i,i+k}. \tag{3.1}$$

WERT(-2, :)	0	0	0	5	10	4	0
WERT(-1, :)	0	3	0	2	0	0	3
WERT(0, :)	5	8	9	1	7	3	12
WERT(1, :)	0	0	0	0	4	0	0
WERT(2, :)	0	6	2	0	0	0	0
WERT(3, :)	4	0	0	5	0	0	0
WERT(4, :)	0	1	0	0	0	0	0

Tabelle 3.4: Darstellung der Matrix A im Compressed Diagonal Storage Format

Bemerkung 3.3. *Das CDS-Format hat natürlich meistens zur Folge, daß Nullen mitgespeichert werden. Es werden sogar (Null-)Elemente im Array gespeichert, für die es keine Entsprechung in der Matrix gibt.*

Speicherbedarf : $DS * n * (p + q + 1)$

3.5 Jagged Diagonal Storage

Dieses Format geht effizienter mit dem Speicher um als das CDS-Format und ist nützlich bei der performanten Implementation von iterativen Lösungsverfahren auf Parallel- und

Vektorrechnern (siehe [Saa96]). Zunächst werden aus der Matrix Nullen entfernt und die verbleibenden Nichtnullelemente nach links geschoben, ähnlich wie im CRS-Format. Danach werden die Zeilen der Matrix der Länge nach absteigend sortiert. Nun wird die so entstandene Matrix Spalte für Spalte hintereinander in einem eindimensionalen Array abgespeichert.

$$\begin{pmatrix} 5 & 0 & 0 & 4 & 0 & 0 & 0 \\ 3 & 8 & 0 & 6 & 0 & 1 & 0 \\ 0 & 0 & 9 & 0 & 2 & 0 & 0 \\ 0 & 5 & 2 & 1 & 0 & 0 & 5 \\ 0 & 0 & 10 & 0 & 7 & 4 & 0 \\ 0 & 0 & 0 & 4 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 12 \end{pmatrix} \longrightarrow \begin{pmatrix} 5 & 4 & & \\ 3 & 8 & 6 & 1 \\ 9 & 2 & & \\ 5 & 2 & 1 & 5 \\ 10 & 7 & 4 & \\ 4 & 3 & & \\ 3 & 12 & & \end{pmatrix} \longrightarrow \begin{pmatrix} 5 & 2 & 1 & 5 \\ 3 & 8 & 6 & 1 \\ 10 & 7 & 4 & \\ 5 & 4 & & \\ 9 & 2 & & \\ 4 & 3 & & \\ 3 & 12 & & \end{pmatrix}$$

Diese Datenstruktur wird *Jagged Diagonals* (gezackte Diagonalen) genannt, da hierbei die abgespeicherten Blöcke eben nicht nur Elemente aus einer Diagonalen enthalten. Die Anzahl njd dieser gezackten Diagonalen ist gleich der größten Zahl von Nichtnullelementen in einer Zeile von A. Ist also nnz_i die Anzahl der Nichtnullelemente in Zeile i, so ist

$$njd = \max_{i=1,\ldots,n} nnz_i. \tag{3.2}$$

Für die Datenstruktur werden gebraucht : ein Gleitkommaarray WERT zur Aufnahme der Matrixeinträge, ein Array SPALTE, der die Spaltenindizes der Werte aufnimmt, ein Array JDANFANG, der angibt, an welcher Position in WERT eine neue gezackte Diagonale beginnt und ein Array, der die Permutation der Zeilen widerspiegelt.

WERT	5	3	10	5	9	4	3	2	8	7	4	2	3	12	1	6	4	5	1

SPALTE	2	3	4	7	1	2	4	6	3	5	6	1	4	3	5	4	6	6	7

JDANFANG	1	8	15	18

PERMUTATION	4	2	5	1	3	6	7

Tabelle 3.5: Darstellung der Matrix A im Jagged Diagonal Storage Format

Ein offensichtlicher Nachteil des Jagged Diagonal Storage liegt jedoch darin, daß das Erstellen dieser Datenstruktur durch das Umsortieren am Anfang recht teuer ist. Ein nachträgliches Einfügen oder gar das sukzessive Aufbauen der Struktur ist nicht möglich.

Speicherbedarf : $DS * nnz + IS * (nnz + njd + n)$.

3.6 Skyline Storage

Das *Skyline Storage* (*SKS*) taucht in der Literatur und in vielen Softwarepaketen in diversen Varianten auf. Zum einen hat man das klassische Skyline Storage Format zur Abspeicherung von Matrizen mit einer bestimmten Gestalt. Zum anderen gibt es das Sparse Skyline Format, das wesentlich universeller einsetzbar ist. Allen Skyline-Formaten ist die Unterteilung in linke untere und rechte obere Dreiecksmatrix gemeinsam.

3.6.1 Symmetric und Non Symmetric Skyline Storage Format

Das ursprüngliche Skyline Storage Format ist für die Speicherung von Matrizen mit einer speziellen Struktur gedacht, die man auch variable Bandmatrizen oder Profilmatrizen nennt. Eine solche ist eine Matrix mit Bandstruktur, die unverhältnismäßig viele Nullen in ihren Bändern enthält. Wenn also p die linke untere und q die rechte obere Bandbreite der Matrix sind, dann ist die Anzahl der Elemente mit $a_{ij} \neq 0$ für $i = j + p, j < i$ bzw. mit $a_{ij} \neq 0$ für $i = j - q, j > i$ sehr viel kleiner als $n \cdot (p + q + 1)$.

Beim CDS würde man also viele Nullen mitspeichern. Das klassische SKS speichert nun die linke untere Hälfte der Matrix zeilenweise - immer vom ersten Nichtnullelement der Zeile bis zum Diagonalelement. Die rechte obere Hälfte wird spaltenweise - vom ersten Nichtnullelement der Spalte bis zum Diagonalelement - abgespeichert. Die Diagonalelemente selbst werden je nach Vereinbarung entweder zu den Zeilen oder Spalten oder gänzlich separat gespeichert.

Ist die Matrix symmetrisch, so speichert man nur den linken unteren Teil - dies wird auch als *Symmetric Skyline Storage* (*SSK*, im Gegensatz zum *Nonsymmetric Skyline Storage*, *NSK*) bezeichnet (siehe [Saa94]).

AL	5	3	8	9	5	2	1	10	0	7	4	0	3	3	12

ALANFANG	1	2	4	5	8	11	14	16

AU	4	6	0	2	0	1	0	0	4	5	0	0

AUANFANG	1	1	1	1	4	6	10	13

Tabelle 3.6: Darstellung der Matrix A im Skyline Storage Format

Im Beispiel sind die Elemente der linken unteren Hälfte und die Diagonalelemente im Array AL untergebracht. Im Array ALANFANG sind die Zeilenanfänge im Array AL eingetragen. Für die Zeilenlänge $l(i)$ der Zeile i in AL gilt stets

$$l(i) = ALANFANG[i + 1] - ALANFANG[i].$$

Daraus läßt sich der Spaltenindex der Elemente aus AL berechnen. Ist $l(i) > 0$, so beginnt die i-te Zeile in der Spalte $i - l(i) + 1$.
Der strikte rechte obere Teil der Matrix steht spaltenweise im Array AU. Im Array AUANFANG ist eingetragen, wo eine neue Spalte beginnt. Die Höhe $h(j)$ der Spalte j beträgt

dabei

$$h(j) = AUANFANG[j+1] - AUANFANG[j].$$

Im Beispiel sind die ersten drei Spalten im strikten rechten oberen Dreieck leer $\Leftrightarrow h(1) = h(2) = h(3) = 0$. Ist $h(j) > 0$, so steht das erste Nichtnullelement der j-ten Spalte in der Zeile mit der Nummer $j - h(j)$. So erhält man die Zeilenindizes der Elemente aus AU. Sei n_{AL} die Anzahl der Elemente aus AL, n_{AU} die der Elemente aus AU, dann gilt :

$$n_{AL} = \sum_{i=1}^{n} l(i) = ALANFANG[n+1] - 1 \tag{3.3}$$

$$n_{AU} = \sum_{j=1}^{n} h(j) = AUANFANG[n+1] - 1. \tag{3.4}$$

Speicherbedarf : $DS * (n_{AL} + n_{AU}) + IS * (2n + 2)$.

3.6.2 Symmetric und Unsymmetric Sparse Skyline Storage Format

Für die Sparse Skyline Storage Formate ist es unerheblich, ob die Matrix eine spezielle Gestalt hat, es kann für alle Arten von dünnbesetzten Matrizen eingesetzt werden. Beim *Unsymmetric Sparse Skyline* (*USS*) Format werden sieben Arrays benötigt : AL, ALSPALTE, ALZEILE für den linken unteren Teil, DIAGONALE für die Diagonalelemente und AU, AUZEILE, AUSPALTE für den rechten oberen Teil.

AL	3	5	2	10	4	3	
ALSPALTE	1	2	3	3	4	6	
ALZEILE	1	1	2	2	4	5	6	7
DIAGONALE	5	8	9	1	7	3	2
AU	4	6	2	1	4	5	
AUZEILE	1	2	3	2	5	4	
AUSPALTE	1	1	1	1	3	4	6	7

Tabelle 3.7: Darstellung der Matrix A im Unsymmetric Sparse Skyline Storage Format

Der strikte rechte obere Teil wird gewissermaßen im Compressed Column Format gespeichert und der strikte linke untere Teil im Compressed Row Format. Auf die Diagonalelemente hat man gesondert Zugriff. Diese Dreiteilung ist bei manchen Algorithmen, wie beispielsweise dem Jakobi- oder dem SOR-Verfahren, hilfreich. Beim *Symmetric Sparse*

Skyline (*SSS*) Format für symmetrische Matrizen speichert man nur den linken unteren Teil und die Diagonale.

Eine Variante des geschilderten Sparse Skyline Formates ist es, den rechten oberen Teil im CRS und den linken unteren im CCS zu speichern, wie es z.B. in der Softwarebibliothek implementiert ist.

Definiert man die Länge einer Zeile aus AL und die Höhe einer Spalte aus AU sinngemäß zum klassischen Skyline Storage, so hat man

$$\begin{aligned} l(i) &= ALZEILE[i+1] - ALZEILE[i] \\ h(j) &= AUSPALTE[j+1] - AUSPALTE[j], \end{aligned}$$

womit sich n_{AL} und n_{AU} wie in (3.3) und (3.4) berechnen.

Speicherbedarf : $DS * (n_{AL} + n_{AU} + n) + IS * (n_{AL} + n_{AU} + 2n + 2)$.

3.7 Matrizen mit Blockstruktur

Bei der Diskretisierung von partiellen Differentialgleichungen entstehen immer dann Matrizen mit Blockstruktur, wenn jeder Gitterpunkt mehrere Freiheitsgrade aufweist. Blockstruktur heißt hier, daß die Matrix aus vielen – in aller Regel quadratischen – Teilmatrizen besteht, die dichtbesetzt sind. Nutzt man diese Eigenschaft aus, so kann man sich nicht nur die Speicherung einiger Indizes sparen, auch kann man einen solchen dichten Teilblock – beispielsweise bei der Matrix-Vektor-Multiplikation – im ganzen bearbeiten, da man sich die Positionsindizes nicht erst beschaffen muß. Außerdem haben viele moderne Prozessoren parallel arbeitende Pipelines, mit denen sich mehrere Rechenoperationen gleichzeitig ausführen lassen. Wenn sich jede Pipeline mit einem Matrixeintrag beschäftigt, läßt sich so eine erhebliche Beschleunigung erzielen.

3.7.1 Block Compressed Row Storage

Das *Block Compressed Row Storage* (*BCRS, BSR*) Format ist eine Erweiterung des CRS Formates. Statt einzelne Gleitkommazahlen werden nun die Matrixblöcke hintereinander gespeichert. Hierbei wird vorausgesetzt, daß alle Blöcke quadratisch sind und dieselbe Dimension haben.

Bezeichne $nnzb$ die Anzahl der Nichtnullblöcke in der Matrix, n_b die Dimension eines jeden Blockes. $n_d := n/n_b$ ist die Blockdimension der Matrix. Dann benötigt man zur Speicherung einen Array der Größe $nnzb * n_b^2$, um die Blöcke aufzunehmen, einen Array der Größe $nnzb$, um die Spaltenindizes der $(1,1)$-Elemente eines jeden Blockes zu speichern und einen Array der Länge $n_d + 1$, der anzeigt,wann eine neue Zeile mit Blöcken beginnt. In [DLH00] wird ein Vergleich zwischen dem CSR- und dem BSR-Format gezogen. Bei entsprechend strukturierten Matrizen zeigt sich bei der Verwendung des BSR-Formates eine erhebliche Speicherersparnis und Reduktion der Rechenzeit, die bei Matrix-Vektor-Operationen bis zu 50% betragen können. Außerdem unterstützt das BSR durch die explizite Aufteilung der Matrix in Blöcke die Parallelisierung von Algorithmen, was in [DLH00] ebenfalls durch numerische Tests belegt wird.

Kapitel 4

Beschreibung der Klassenbibliothek

adr ist ein Finite-Elemente-Programm zur Lösung von Konvektions-Diffusions-Reaktions-Gleichungen, das am Institut für Numerische und Angewandte Mathematik der Universität Göttingen entwickelt wird. Es besteht aus verschiedenen Bibliotheken, von denen in diesem Kapitel nun mit **lins** (*library of iterative numerical solvers*) diejenige beschrieben wird, die sich mit der Darstellung und Lösung von Gleichungssystemen befaßt. Die Illustration und Dokumentation der Klassenbibliotheken geschieht, wo es sich anbietet, in der Notation der Modellierungssprache *UML* (*Unified Modelling Language*, siehe z.B. [BRJ99]).

4.1 Einführung

Wie im Laufe von Kapitel 1 demonstriert wurde, werden (elliptische) partielle Differentialgleichungen wie beispielsweise (1.20) in der Praxis dadurch numerisch behandelt, daß man sie in geeigneter Weise diskretisiert (z.B. mit Finiten Elementen) und dann das so entstehende lineare Gleichungssystem löst. Eine Software, die solche Berechnungen durchführen kann, wird als *FEM-Programm* bezeichnet.

4.1.1 adr : ein FEM-Programm

Wie im UML-Diagramm in Abbildung 4.1 dargestellt wird, benötigt eine solche FEM-Software zur Berechnung der diskreten Lösung im wesentlichen die folgenden Funktionen:

1. Eingabe der Problembeschreibung,
2. Generierung des Netzes und eventuelle Verfeinerung,
3. Assemblierung der Steifigkeitsmatrix und der rechten Seite des linearen Gleichungssystems aus der diskretisierten Variationsgleichung und Einarbeiten der Randbedingungen,
4. Lösung des linearen Gleichungssystems durch geeignete (direkte oder iterative) Verfahren,
5. Aufbereitung und Ausgabe der Lösung.

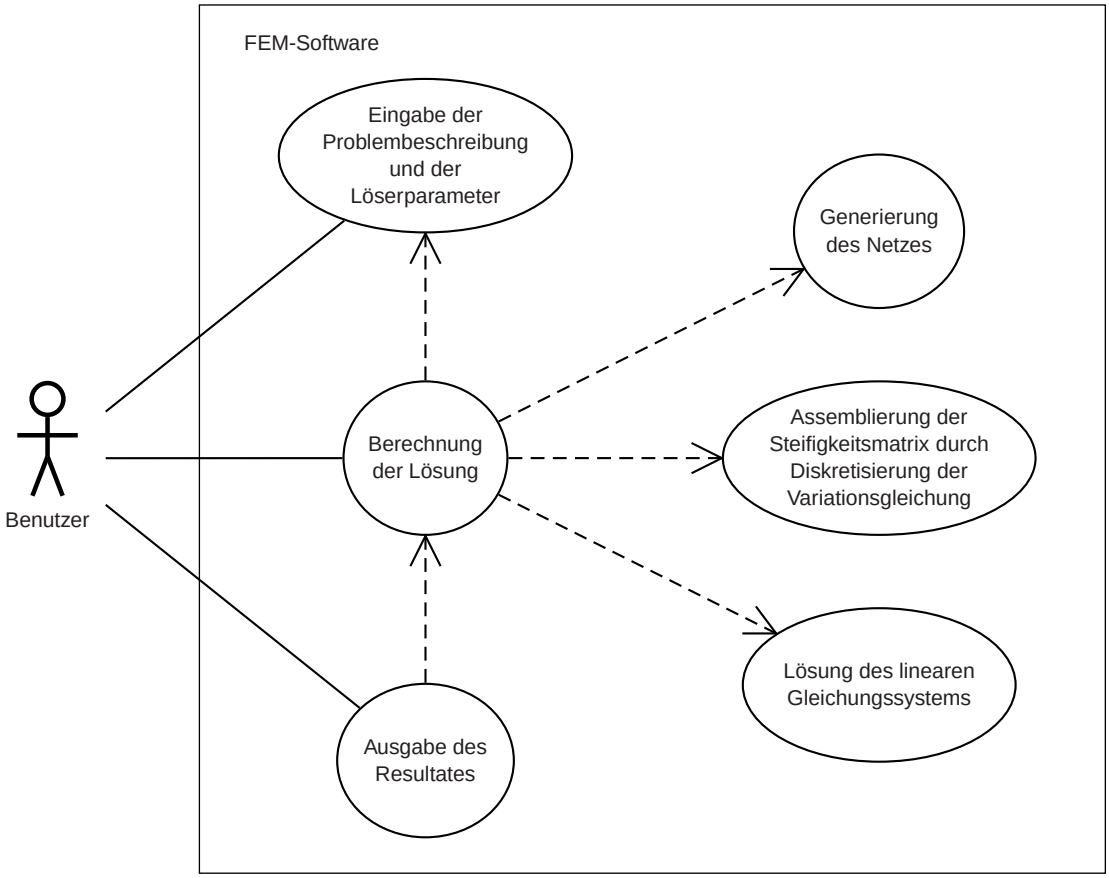


Abbildung 4.1: Anwendungsfalldiagramm einer FEM-Software

Die Netzgenerierung und die Art ihrer Implementierung in **adr** wird ausführlich in [Koh00] beschrieben. Die Dokumentation der Diskretisierung in der jetzigen Form steht hingegen noch aus.

Alle diese (internen) Anwendungsfälle muß jede FEM-Software in irgendeiner Form zur Verfügung stellen : Ohne ein Netz kann keine Diskretisierung erfolgen, und ohne eine durch die Diskretisierung erzeugte Matrix kann kein Gleichungssystem gelöst werden, so daß die zeitliche Reihenfolge vorgegeben ist. Es kann jedoch durch verschiedene Zusatzfunktionen wie Adaptivität, Mehrgitterverfahren oder Shock-Capturing über diese Arbeitsschritte iteriert werden. Stellt man z.B. durch geeignete Fehlerschätzer fest, daß in manchen Teilen des Gebietes der Fehler zu groß wird, so wird das Netz dort entsprechend verfeinert. Dazu muß der Teil des Programms, der für die Netzverfeinerung zuständig ist, erneut aufgerufen werden. Im Diagramm 4.1 ist die Methode der Gebietszerlegung noch nicht vorgesehen, sie wird in einer kommenden Version von **adr** integriert werden.

adr ist dazu gedacht, das bisher verwendete Programm $\mathcal{PNS}$ (vgl. [AO$^+$99]) zu ersetzen. Die Zielsetzungen waren hierbei hauptsächlich :

- Verbesserte Handhabung und Bedienung :
  Beispielsweise muß **adr** nicht mehr für jedes Problem neu übersetzt werden, ganz im Gegensatz zu $\mathcal{PNS}$, wo die Problemdaten im Programmcode fest verankert war.

- Leichtere Erweiterbarkeit und Wartbarkeit :
  Durch die durchgehend objektorientierte Struktur sind Konzepte wie Adaptivität, Gebietszerlegung, Parallelisierung schneller zu integrieren.

- Erhöhte Flexibilität und Funktionalität :
  In **adr** sind bereits neuartige Verfahren wie Downwind-Numbering (vgl. [Sch00]) und mehrstufige Vorkonditionierung implementiert, weitere sind in Vorbereitung.

4.1.2 Anforderungen und Zielsetzungen

Es war ein wesentliches Ziel dieser Arbeit, eine objektorientierte Klassenbibliothek für **adr** zu entwickeln, die die Darstellung und numerische Lösung von Gleichungssystemen erlaubt. Sie sollte gewisse Datenstrukturen zur Verfügung stellen, mit denen sich die bei der Diskretisierung entstehenden, dünnbesetzten Matrizen speichern lassen. Außerdem sollte sie einen Pool von robusten, iterativen Lösungsverfahren enthalten, wie sie in Kapitel 2 vorgestellt wurden.

Genauer wurden an die *Datenstrukturen* folgende Anforderungen gestellt :

1. Es sollten Datenformate für dichtbesetzte und für dünnbesetzte Matrizen bereitgestellt werden, die sich leicht mit modernen Iterationsverfahren einsetzen lassen.

2. Das Design der Datenstrukturen sollte nicht nur dünnbesetzte Matrizen mit skalaren Einträgen und dünnbesetzte Matrizen, deren Einträge wiederum kleine dichtbesetzte Blockmatrizen sind (wie in Abschnitt 3.7 beschrieben), berücksichtigen, sondern darüber hinaus eine Verallgemeinerung dieses Konzeptes anbieten. Hiermit sind ganz allgemein *„verschachtelte Matrizen“* gemeint, also Matrizen, die als Einträge wiederum Matrizen haben, deren Einträge ebenfalls Matrizen sind, und (theoretisch) so weiter, ohne daß genauere Anforderungen an deren konkrete Gestalt gestellt werden.

3. Generell sollte die Beschränkung auf quadratische Matrizen aufgehoben werden. Es sollten also auch (insbesondere für Teilmatrizen) rechteckige Matrizen zugelassen sein.

Diese Forderungen resultieren hauptsächlich aus der zukünftigen Ausrichtung von **adr** auf Gebietszerlegungsverfahren und Schur-Komplement-Methoden, wobei die auftretenden Teilmatrizen verwaltet werden müssen. Dies war mit blanc (siehe [Pri96]), der bei $\mathcal{PNS}$ bisher verwendeten Matrixbibliothek, nicht in dieser Form möglich.

Die Menge der *Lösungsverfahren* sollte folgende Eigenschaften haben :

1. Es sollten Verfahren für symmetrische und nichtsymmetrische Probleme zur Verfügung stehen. Neben den Krylov-Verfahren sollten auch einige Splitting-Verfahren

enthalten sein, damit sich deren Verhalten bei der Kombination mit Downwind-Numbering untersuchen läßt. Auch sollen geeignete Varianten der Splitting-Methoden als Vorkonditionierer fungieren.

2. Alle Lösungsverfahren sollten prinzipiell die Links- und Rechts-Vorkonditionierung erlauben. Insbesondere sollten Vorkonditionierer wieder vorkonditionierbar sein, so daß sich eine mehrstufige Vorkonditionierung wie im Falle des SPD-Vorkonditionierers (siehe Abschnitt 2.4.6) realisieren und testen läßt.

Desweiteren sollten alle Komponenten von **adr** in C++ implementiert werden, um einerseits die eventuelle Einbindung von C-Bibliotheken zu ermöglichen und andererseits die Vorteile einer objektorientierten Sprache auszunutzen. Außerdem zeigen verschiedene Präzedenzfälle wie *Blitz++* (vgl. [Vel99]) oder die *MTL* (vgl. [Sie99]), daß C++ in Fragen der Laufzeit und Performance keinen Vergleich mit Fortran oder C scheuen muß (siehe z.B. auch [VJ97]).

Das Design von **adr** wurde nach modernen, objektorientierten Kriterien entworfen, wie sie etwa in [Str98] oder [Swi99] beschrieben sind. Auf die Konzepte der Matrix- und der Löserbibliothek wird im folgenden eingegangen.

4.2 Container

Die Matrizen werden in dieser Bibliothek durch abstrakte Datentypen modelliert und entsprechend dem objektorientierten Paradigma durch Klassen realisiert. Es handelt sich hierbei insbesondere um *generische Klassen*, denn sie erhalten einen Parameter, der den Typ der Matrixeinträge angibt. In der Sprache der Standardbibliothek von C++ (*Standard Template Library, STL*, siehe [Str98], Teil III) werden derartige Klassen auch als *Container* bezeichnet, andernorts werden sie auch *Träger* genannt. In C++ verwirklicht man generische bzw. parametrisierte Klassen durch *Templates*.

Die allgemeine Basisklasse für Matrizen heißt `Matrix<T>`, wobei `T` der Template-Parameter ist. Von ihr sind alle weiteren Matrix- und Vektorklassen dieser Bibliothek abgeleitet. In Abbildung 4.2 werden zunächst die grundlegenden Attribute und Operationen dieser Klasse dargestellt. Im Verlauf des Kapitels werden - je nach betrachtetem Aspekt - weitere vorgestellt.

Die Klasse `Matrix<T>` ist *abstrakt* (oder *virtuell* im Sprachgebrauch von C++), d.h. von ihr können keine Objekte instanziiert werden. Dies ist auch nicht erforderlich, da sie die allgemeinste Abstraktion einer Matrix darstellt. Aus diesem Grund haben die Konstruktoren auch nur den Zweck, die entsprechenden Attribute wie `theRows`, `theColumns` und `symmetric` zu setzen. Mit den Funktionen `rows()` und `columns()` kann die Zeilen- bzw. Spaltenzahl abgefragt werden.

4.2.1 Kapselung und Varianten von Matrizen

Ein wesentliches Designmerkmal einer solchen Klassenbibliothek ist es, daß die Matrizen vollständig gekapselt sind, d.h. eine Matrix bietet nach außen eine Menge von Operatio-

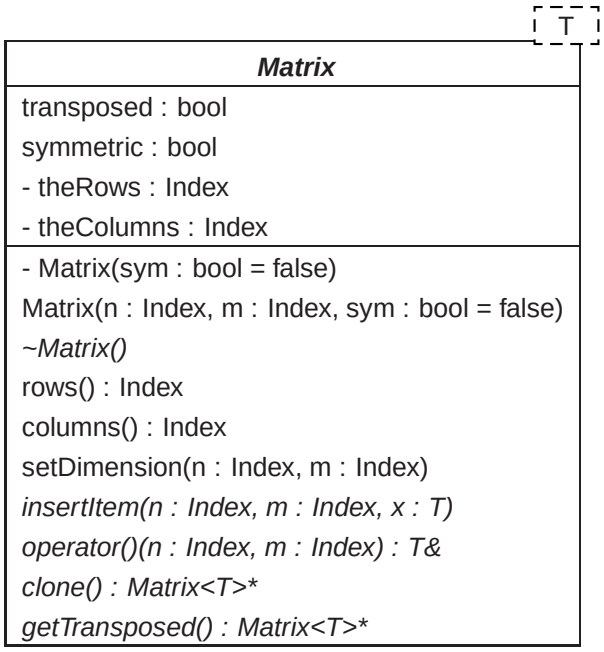


Abbildung 4.2: Die abstrakte Klasse `Matrix<T>`

nen an und verrät nichts über die interne Implementierung.

In vielen Fällen benötigt man spezielle Varianten von Matrizen - symmetrische, transponierte oder inverse Matrizen. In **lins** werden diese Varianten folgendermaßen modelliert:

- *Symmetrische Matrizen* werden durch gewöhnliche Matrix-Objekte repräsentiert, bei denen lediglich das `symmetric`-Flag auf `true` gesetzt wurde. Ein Matrix-Objekt kennt diesbezüglich also nur zwei Zustände: Entweder ist es symmetrisch oder nicht. Wesentlich ist hierbei, daß im symmetrischen Fall nur die obere rechte Hälfte (genauer gesagt: die strikte rechte obere Dreiecksmatrix und die Diagonale) gespeichert wird. Wird auf die untere Hälfte zugegriffen, so wird der entsprechende Wert aus der oberen Hälfte zurückgeliefert. Algorithmen können, zugunsten der Effizienz, das `symmetric`-Flag abprüfen, jedoch ist das semantisch korrekte Funktionieren der Algorithmen auch dann gewährleistet, wenn dies nicht geschieht. Der Vorteil der gesonderten Behandlung von symmetrischen Matrizen besteht also in der Speicherersparnis und im Laufzeitverhalten gewisser Algorithmen (siehe Abschnit 4.6.1).

- *Transponierte Matrizen* sind – im Gegensatz zu symmetrischen – einer anderen Matrix zugeordnet. Sie werden, um Zeit und Speicherplatz zu sparen, erst auf Anforderung mit der Funktion `getTransposed()` erzeugt. Dabei handelt es sich um *Dummy-Objekte*, deren Funktionsweise weiter unten erläutert wird.

- *Inverse Matrizen* kann es ebenfalls zu jeder Matrix geben; man erhält mit der Funktion `getInverse()` einen Zeiger auf ein Objekt, welches die inverse Matrix modelliert. Hier sind jedoch zwei Fälle zu unterscheiden. Einerseits ist es möglich, daß es sich dabei um ein gewöhnliches Matrix-Objekt (mit physikalisch existierenden Matrixeinträgen) handelt, die beispielsweise durch ein Gaußsches Eliminati-

onsverfahren erzeugt wurde. Hat man eine solche Matrix vorliegen, so erlaubt es die Funktion `setInverse(Matrix<T>*)`, die Ausgangsmatrix mit ihrer Inversen zu verknüpfen. Andererseits ist es auch möglich, daß es sich bei der Inversen wieder nur um ein Dummy-Objekt handelt, das durch den entsprechend gesetzten `theInverter`-Zeiger weiß, wie es sich beispielsweise bei der Multiplikation mit einem Vektor verhalten muß. Direkte Zugriffe auf einzelne Matrixeinträge sind dann natürlich nicht möglich und werden durch eine Exception abgefangen. Abbildung 4.3 zeigt die Assoziation zwischen dem Matrix- und dem `theInverter`-Objekt, das vom Typ `Solver<Matrix<T>>` ist (siehe Abschnitt 4.3.2) und durch die Funktion `setInverter(Solver<Matrix<T>>*)` gesetzt wird.

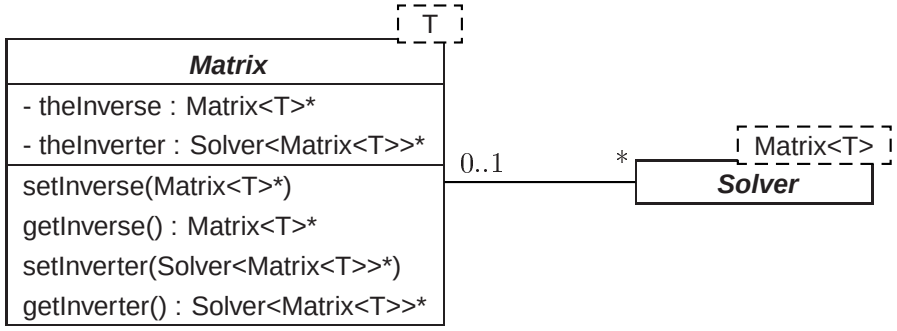


Abbildung 4.3: Assoziationen und Operationen zur Modellierung von Inversen Matrizen

Bemerkung 4.1. *Für eine große dünnbesetzte Matrix wird natürlich keine echte inverse Matrix angelegt, sondern nur ein Dummy-Objekt. Jedoch kann es sich für Matrizen mit kleinen dichtbesetzten Blöcken durchaus lohnen, zumindest die Blockmatrizen auf der Diagonalen echt zu invertieren. Auf diese Weise kann man dann beispielsweise bei einem Algorithmus zum Rückwärtseinsetzen (oder beim Jacobi-Verfahren, etc.) direkt die inverse Blockmatrix mit einem entsprechend großen Teilvektor multiplizieren.*

Unter *Dummy-Objekten* sind hierbei Objekte zu verstehen, die keine eigenen Daten enthalten, sondern nur Verweise auf die eigentlichen Daten (genauer : die Matrixeinträge) und darüber hinaus nur die Information, wie mit ihnen umgegangen werden soll. Eine „echte“ Matrix verfügt hingegen über alle Daten selbst, und wenn sie erzeugt wird, wird entsprechender Speicher bereitgestellt. Eine Dummy-Matrix wird jedoch erzeugt, ohne Speicher für die Matrix-Elemente zu reservieren. Dies geschieht mit dem leeren Konstruktor, d.h. einem Konstruktor wie `Matrix(sym : bool = false)`, der keine Dimensionsangabe verlangt. Nach außen ist dieser Unterschied jedoch nicht sichtbar, das Verhalten der Objekte ist für den Aufrufer völlig einheitlich.

Die Funktion `getTransposed()` liefert beispielsweise einen Zeiger auf die transponierte Matrix. Beim ersten Aufruf dieser Funktion wird eine leere Matrix (mit vertauschten Rollen für `theRows` und `theColumns`) erzeugt, und die Zeiger auf die Daten werden entsprechend gesetzt. Zusätzlich wird das Flag `transposed` auf `true` gesetzt, damit das Objekt weiß, auf welche Weise auf die Daten zugegriffen werden muß. Sämtliche Funktionen, die dieses Wissen benötigen, müssen dieses Flag prüfen, bevor sie auf die Daten der Matrixeinträge zugreifen.

Bemerkung 4.2. *Obwohl die Funktion* `getTransposed()` *in der Klasse* `Matrix<T>` *eine rein virtuelle* (pure virtual) *Funktion ist, d.h. die Realisierung erst in den abgeleiteten Klassen geschieht, ist das oben beschriebene Prinzip jedoch in jeder dieser Klassen verwirklicht.*

4.3 Die Klassenhierarchie

Die Klasse `Matrix<T>` ist, genauso wie die Klasse `Solver<MatrixT, VectorT>`, die in Abschnitt 4.3.2 ausführlich vorgestellt wird, abstrakt, d.h. sie dient hauptsächlich dazu, eine allgemein verläßliche Schnittstelle zu definieren, die für alle Matrizen (bzw. für alle Löser) gleich ist. Konkrete Implementationen werden erst durch Unterklassen gegeben. Diese Klassen, ihre Vererbungshierarchie und ihre Abhängigkeiten werden in den nächsten beiden Unterabschnitten beschrieben. Anschließend wird das Konzept der Abbruchkontrolle für die Lösungsverfahren und seine Implementierung vorgestellt.

4.3.1 Matrix- und Vektorklassen

In Kapitel 3 wurden verschiedene Speichertechniken für dünnbesetzte Matrizen dargestellt. Jeder dieser Techniken liegt eine abstrakte Sichtweise auf die Matrix zugrunde, die sich auch auf dichtbesetzte Matrizen anwenden läßt. Hiermit lassen sich die Datenstrukturen danach einordnen, ob sie die Einträge

1. zeilenweise (z.B. CRS),
2. spaltenweise (CCS),
3. bandweise (CDS),
4. nach oberer und unterer Dreiecksmatrix getrennt (SKS) oder
5. nach oberer, unterer Dreiecksmatrix und Diagonale getrennt (SSS, USS)

im Speicher ablegen. Um diese Sichtweisen zu berücksichtigen, wurde in der Klassenhierarchie eine Abstraktionsebene zwischen der Klasse `Matrix<T>` und den konkreten Implementationen eingefügt. Dies sind die abstrakten Klassen `RowWiseMatrix<T>` und `ButterflyMatrix<T>`. Klassen, die ihre Elemente zeilenweise abspeichern (wie in 1.), erben von `RowWiseMatrix<T>`, und jene, die ihre Daten nach oberer und unterer Dreiecksmatrix und Diagonale getrennt (wie in 5.) speichern, erben von `ButterflyMatrix<T>`.

Bemerkung 4.3. *Für die Speichertechniken 2.,3. und 4. gibt es derartige Oberklassen nicht, sie sind derzeit nicht implementiert. Es ist auch davon auszugehen, daß spaltenweise Speicherung beispielweise gegenüber einer zeilenweisen keine Verbesserung erbringt. Auch unterscheidet sich die vierte Variante nur unwesentlich von der fünften. Lediglich die bandweise Abspeicherung verspricht bei entsprechend strukturierten Matrizen einen Performancegewinn und wird in einer späteren Version evtl. noch implementiert werden.*

Die Klasse `ButterflyMatrix<T>` hat als Besonderheit drei Funktionen, die als Resultat einen Zeiger auf den rechten oberen Teil (`getUpperPart()`), auf die Diagonale

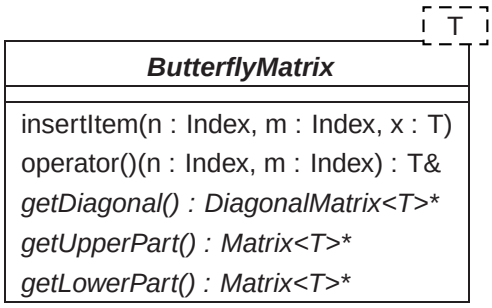


Abbildung 4.4: Die abstrakte Klasse `ButterflyMatrix<T>`

Klasse	**Eigenschaft**
`DiagonalMatrix<T>`	$a_{ij} = 0$ für $i \neq j$
`TriangularSparseMatrix<T>`	A ist dünnbesetzt und $a_{ij} = 0$ für $i \geq j$
`TriangularDenseMatrix<T>`	A ist dichtbesetzt und $a_{ij} = 0$ für $i \geq j$
`DenseVector<T>`	A ist dichtbesetzter Vektor, d.h. $m = 1$

Tabelle 4.1: Unterklassen von `RowWiseMatrix<T>`

(`getDiagonal()`) bzw. auf den linken unteren Teil (`getLowerPart()`) liefern (siehe Abbildung 4.4). Die Funktionen `insertItem` und `operator()` können auf dieser Abstraktionsebene bereits implementiert werden, da sie lediglich - in Abhängigkeit von den übergebenen Indizes - ihre Parameter an die entsprechenden Teilmatrizen weiterleiten.

Die Unterklassen von `RowWiseMatrix<T>` und ihre Einsatzgebiete sind in Tabelle 4.1 für eine Matrix $A \in \mathbf{R}^{n \times m}$, $A = (a_{ij}), i = 1, \ldots, n, j = 1, \ldots, m$, dargestellt, die Spezialisierungen von `ButterflyMatrix<T>` entnimmt man Tabelle 4.2.

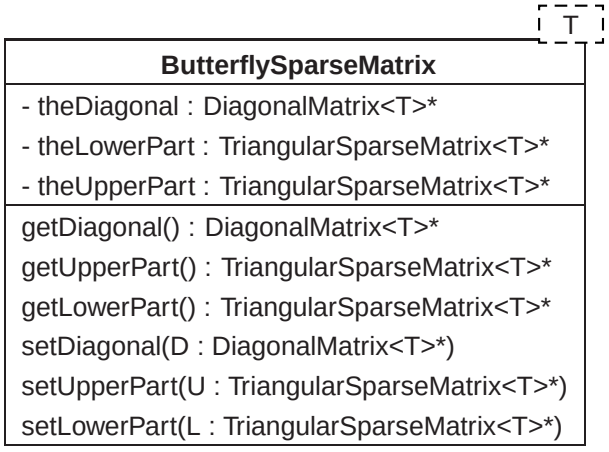


Abbildung 4.5: Die Klasse `ButterflySparseMatrix<T>`

Im Klassendiagramm in Abbildung A.1 findet man den gesamten Vererbungsbaum und alle Abhängigkeiten der Matrizenklassen untereinander.

Klasse	Eigenschaft
`ButterflySparseMatrix<T>`	A ist dünnbesetzt
`ButterflyDenseMatrix<T>`	A ist dichtbesetzt

Tabelle 4.2: Unterklassen von `ButterflyMatrix<T>`

Die Klasse `ButterflySparseMatrix<T>` implementiert die rein virtuellen Funktionen von `ButterflyMatrix<T>` durch Verwendung des Typs `TriangularSparseMatrix<T>` (vgl. Abb. 4.5). Darüber hinaus bietet sie verschiedene Funktionen zum Setzen der entsprechenden Teilmatrizen.

4.3.2 Löser und Vorkonditionierer

Unter einem *Löser* im Sinne der Klassenbibliothek ist ein Objekt zu verstehen, welches ein Verfahren anbietet, das zu einem gegebenen Gleichungssystem $Ax = b$ und einer Anfangsnäherung x_0 in einer endlichen Anzahl von Schritten eine „bessere“ Näherung x_1 generiert. Legt man diese Sichtweise zugrunde, so kann man auch Vorkonditionierer als Löser im Sinne der Klassenbibliothek ansehen, wenn man sich in Erinnerung ruft, daß eine Vorkonditionierungsmatrix in der Regel eine Approximation der Inversen von A darstellt (vgl. Abschnitt 2.4).

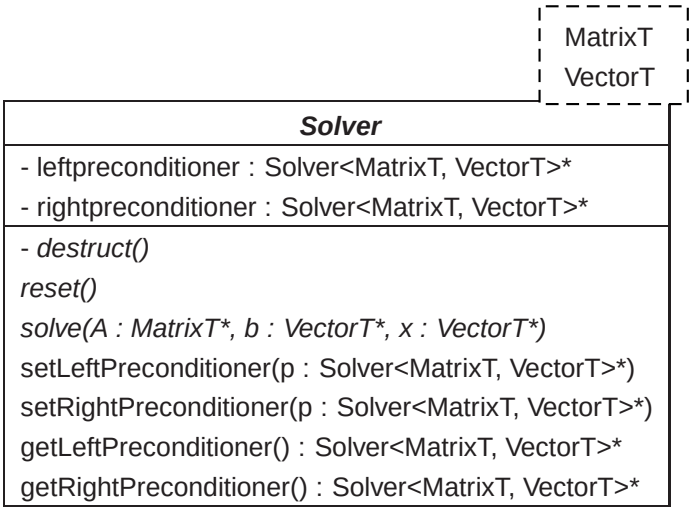


Abbildung 4.6: Die Klasse `Solver<MatrixT, VectorT>`

Dieses abstrakte Konzept der Löser wird in der Klassenbibliothek durch die Klasse `Solver<MatrixT, VectorT>` (siehe Abb. 4.6) modelliert. Auch diese Template-Klasse ist abstrakt. Sie erhält zwei Parameter, die den Typ der verwendeten Matrixklasse bzw. Vektorklasse angeben. Im wesentlichen enthält `Solver<MatrixT, VectorT>` nun die Methode

- `solve(A : MatrixT*, b : VectorT*, x : VectorT*)`,

die zu einem gegebenen Zeiger auf eine Matrix `A`, einem Zeiger auf die rechte Seite `b` und auf die Startnäherung `x` eine Näherungslösung berechnet. Nach der Berechnung zeigt `x` auf diese neue Approximation.

Die Konzeption sieht vor, daß jedes Lösungsverfahren durch eine eigene Klasse realisiert wird und nicht etwa beispielsweise durch eine Template-Funktion. Eine solche Klasse erbt nun von `Solver<MatrixT, VectorT>` und implementiert die Methode `solve`. Ein Löserobjekt ist also ein „Experte“ für eine bestimmte Art von Berechnung. Für die Vorkonditionierer wurden eigene Klassen geschrieben, da sie in der Regel weniger temporäre Objekte brauchen. Auch führen sie ja auch nur einen Iterationsschritt durch, so daß sich hier Vereinfachungen im Code ergeben. Will man in einem Iterationsschritt des eigentlichen Lösers den Vorkonditionierer öfter anwenden, so muß man jedoch die entsprechende Löserklassen verwenden.

In Tabelle 4.3 bzw. 4.4 sind sämtliche Lösungsverfahren bzw. Vorkonditionierer und die Klassen, durch die sie modelliert werden, gegenübergestellt.

Weiterhin ermöglichen es nun die Methoden

- `setLeftPreconditioner(p : Solver<MatrixT, VectorT>*)` und
- `setRightPreconditioner(p : Solver<MatrixT, VectorT>*)`,

jedem Löserobjekt einen Vorkonditionierer zuzuordnen, wodurch sich die Möglichkeit der mehrstufigen Vorkonditionierung ergibt. Der Verschachtelungstiefe sind hierbei keine Grenzen gesetzt. Erst wenn die Funktionen `getLeftPreconditioner` bzw. `getRightPreconditioner` den Nullzeiger als Ergebnis zurückliefern, bricht die Kette der Vorkonditionierer ab. In der Praxis macht es jedoch nur in gewissen Fällen Sinn, einen Vorkonditionierer wieder vorzukonditionieren.

Bemerkung 4.4. *Die Verfahren von Zeile 6 bis 10 in Tabelle 4.3 (also Gauß-Seidel bis Rückwärtseinsetzen) sowie die Verfahren aus Zeile 3 bis 7 in Tabelle 4.4 können nur dann ordnungsgemäß funktionieren, wenn die Parameterklasse* `MatrixT` konform *zu* `ButterflyMatrix<T>` *ist, d.h. wenn sie von* `ButterflyMatrix<T>` *erbt. In C++ gibt es, im Gegensatz zu EIFFEL, leider kein Sprachelement, um dies bereits im Programmcode auszudrücken. Die Löser in den ersten beiden Zeilen arbeiten dagegen mit jeder Art von (nicht-virtuellen) Matrixklassen zusammen, die zu* `Matrix<T>` *konform ist. Eine Sonderstellung nehmen die Löserklassen* `JacobiSolver` *und* `JORSolver` *(und analog die Vorkonditioniererklassen* `JacobiPreconditioner` *und* `JORPreconditioner`*) ein. Sie benötigen nicht unbedingt eine zu* `ButterflyMatrix<T>` *konforme Klasse als Parameter, sondern lediglich eine Klasse, die die Diagonalelemente separat in einer Diagonalmatrix abspeichert, wie es z.B. beim Modified Row Storage (siehe Abschnitt 3.2) der Fall ist.*

In [Hac93], Unterkapitel 4.5 beispielsweise, oder in [BBC+94], Abschnitt 3.2.1, werden spezielle Block-Varianten der Splitting-Verfahren vorgestellt. Hierbei unterteilt man die Matrix in kleinere (quadratische) Blöcke und formuliert den Algorithmus dann statt auf den einzelnen skalaren Matrixeinträgen auf eben diesen Blockmatrizen. In der vorliegenden Klassenbibliothek existieren jedoch keine gesonderten Block-Varianten der Algorithmen, denn jedes Lösungsverfahren ist automatisch dann ein Block-Verfahren, wenn Blockmatrizen verwendet werden (vgl. Bemerkung 4.1 und Abschnitt 4.4). Dies ist eine Konsequenz

Lösungsverfahren	Implementierende Klasse
Konjugierte Gradienten	`CGSolver<MatrixT,VectorT>`
GMRES	`GMRESSolver<MatrixT,VectorT>`
BiCGSTAB	`BiCGSTABSolver<MatrixT,VectorT>`
Jacobi	`JacobiSolver<MatrixT,DiagonalT,VectorT>`
JOR	`JORSolver<MatrixT,DiagonalT, TriangularT,VectorT>`
Gauß-Seidel	`GaussSeidelSolver<MatrixT,DiagonalT, TriangularT,VectorT>`
SOR	`SORSolver<MatrixT,DiagonalT, TriangularT,VectorT>`
SSOR	`SSORSolver<MatrixT,DiagonalT, TriangularT,VectorT>`
Vorwärtseinsetzen	`ForwardInserter<MatrixT,DiagonalT, TriangularT,VectorT>`
Rückwärtseinsetzen	`BackwardInserter<MatrixT,DiagonalT, TriangularT,VectorT>`

Tabelle 4.3: Löserklassen

aus der Tatsache, daß alle Löser in der Klassenbibliothek (außer `ForwardInserter` und `BackwardInserter`) auf einem hohen Abstraktionsniveau implementiert sind. Sie verwenden lediglich mathematische Operationen, wie Matrix-Vektor-Multiplikation, Addition und Skalierung von Vektoren, und greifen dabei *nicht* (im Gegensatz zu vielen anderen Bibliotheken), auf die einzelnen Matrixelemente zu. Dadurch werden die *Löser* (Algorithmen) weitestgehend von den *Matrizen* (Datenstrukturen) entkoppelt, was ein wichtiger Aspekt des objektorienterten Designs ist.

Die Verfahren des Vorwärts- und Rückwärtseinsetzens sind die einzigen, die einen etwas tieferen Einblick in die Matrixstruktur haben müssen. Sie erhalten diesen über (zweidimensionale) Iteratoren (vgl. Abschnitt 4.5). Wie auch anhand der Abhängigkeiten in den Klassendiagrammen A.3 bis A.5 in Anhang A zu sehen ist, wird das Prinzip des Vorwärts- bzw. Rückwärtseinsetzens von vielen anderen Verfahren benutzt. Aus diesem Grund wurden sie dort herausgenommen und als separate Verfahren in Gestalt der Klassen `ForwardInserter` und `BackwardInserter` implementiert.

In Abbildung A.2 in Anhang A sieht man die Klassen, welche die Krylov-Unterraum-Methoden implementieren. Hieran soll exemplarisch die Speicherverwaltung der Löserklassen erläutert werden. Man sieht zunächst die private Funktion `init`, die in jedem Löser die Aufgabe hat, den Speicher für die temporären Variablen und Datenstrukturen, die bei der eigentlichen Berechnung gebraucht werden, bereitzustellen und gegebenenfalls zu initialisieren. Da die Speicherbeschaffung verhältnismäßig zeitaufwendig ist, wird `init` einmal am Anfang von `solve` aufgerufen, und zwar genau dann, wenn das Flag `initialized` den Wert `false` hat. Dies ist z.B. unmittelbar nach dem Erzeugen des Löserobjektes der

Vorkonditionierungs-verfahren	**Implementierende Klasse**
Jacobi	`JacobiPreconditioner<MatrixT,DiagonalT, VectorT>`
JOR	`JORPreconditioner<MatrixT,DiagonalT, VectorT>`
SOR	`SORPreconditioner<MatrixT,DiagonalT, TriangularT,VectorT>`
SSOR	`SSORPreconditioner<MatrixT,DiagonalT, TriangularT,VectorT>`
Unvollständige LU-Zerlegung	`ILUPreconditioner<MatrixT,DiagonalT, TriangularT,VectorT>`
Unvollständige Cholesky-Zerlegung	`ICPreconditioner<MatrixT,DiagonalT, TriangularT,VectorT>`
SPD	`SPDPreconditioner<MatrixT,DiagonalT, TriangularT,VectorT>`

Tabelle 4.4: Vorkonditioniererklassen

Fall. Nach getaner Arbeit setzt die Funktion `init` dann die Variable `initialized` auf `true`, so daß sie von `solve` nicht mehr ohne weiteres aufgerufen werden kann. Erst, wenn die Funktion `reset` aufgerufen wird, hat `initialized` wieder den Wert `false`. Ist ein Löser nämlich erst einmal initialisiert, so läßt er sich beispielsweise nur mit Matrizen einer festgelegten Größe benutzen. Will man ihn mit einer anderen Matrixgröße verwenden, so muß man zunächst eben `reset` aufrufen, bevor man ihn wieder verwenden kann. Hierbei werden die internen Variablen des Lösers erst beim erneuten Aufruf von `solve` bzw. `init` durch die private Funktion `destruct` freigegeben.

Desweiteren ist in Abbildung A.2 in der Klasse `GMRESSolver` die private Funktion `restartsolve` aufgeführt, die in Abhängigkeit vom `ConvergenceControl`-Objekt (siehe Abschnitt 4.3.4) von der Funktion `solve` aufgerufen wird, um einen GMRES mit Restart (vgl. Algorithmus 2.10) anstelle eines normalen GMRES (vgl. Algorithmus 2.9) durchzuführen.

Genauer passiert folgendes : Der Algorithmus fragt seine Parameter über das ihm mitgegebene `SolverParameter`-Objekt ab (siehe Abschnitt 4.3.3) – dies ist in diesem Fall nur die Dimension des Krylov-Unterraums, und die Abfrage geschieht durch Aufruf der Funktion `getSubspaceDimension()`. Handelt es sich nun beim `ConvergenceControl`-Objekt des Lösers (siehe Abschnitt 4.3.4) um eines vom Typ `CountControl` mit maximaler Iterationszahl 1, so wird das normale GMRES-Verfahren ohne Restart einmal durchgeführt. Andernfalls werden so viele Restarts durchgeführt, wie es das `ConvergenceControl`-Objekt vorschreibt.

4.3.3 Löserparameter

Verschiedene Löser brauchen teilweise diverse Parameter, um ihre Berechnungen durchführen zu können. Das GMRES-Verfahren benötigt eine natürliche Zahl, die die Dimension des Unterraums angibt, das SOR-Verfahren bekommt eine Gleitkommazahl zwischen 0 und 2, welche die Relaxation bestimmt. In der Klassenbibliothek **lins** gibt es dafür die Klasse `SolverParameter`, die wie eine zentrale Datenbank für alle Löser funktioniert. Dort werden alle diese Daten gespeichert, und jeder Löser verfügt über einen Zeiger auf ein solches `SolverParameter`-Objekt, so daß er sich bei Bedarf die für ihn wichtige Information beschaffen kann. Abbildung 4.7 zeigt die Assoziation zwischen den Klassen `Solver` und `SolverParameter`.

Natürlich ist es auch möglich, mehrere verschiedene `SolverParameter`-Objekte zu erzeugen, und sie verschiedenen Lösern zuzuweisen. Dies kann für hybride, mehrstufige Lösungsverfahren, wie sie mit diesen Klassen möglich sind, und die sich aus mehreren Standardlösern zusammensetzen, durchaus sinnvoll sein.

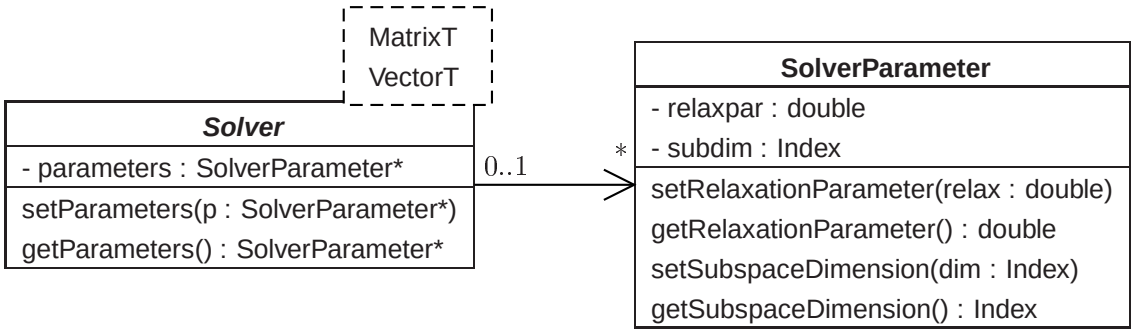


Abbildung 4.7: Löser und Parameter

4.3.4 Konvergenzkontrolle

Um die Abbruchbedingungen für die Löser zu formulieren, gibt es die abstrakte Klasse `ConvergenceControl<VectorT>`. Jedes Löserobjekt besitzt einen Zeiger auf ein Objekt dieser Klasse bzw. einer ihrer Unterklassen. Die Klasse `ConvergenceControl<VectorT>` besitzt als wichtigstes Merkmal die Funktion `converged`, die vom Löser in jedem Schritt aufgerufen werden muß, um festzustellen, ob die aktuelle Näherung einem gewissen Abbruchkriterium genügt. Ist dies der Fall, liefert sie `true` zurück, ansonsten `false`.

Ein Nebeneffekt hiervon ist, daß das `ConvergenceControl<VectorT>`-Objekt dabei die Iterationsschritte mitzählt, so daß sich, nachdem `solve` terminiert ist, mit der Funktion `getLastIterationStep` die Gesamtzahl der benötigten Iterationen erfragen läßt.

Die unterschiedlichen Abbruchkriterien werden durch verschiedene Unterklassen von `ConvergenceControl<VectorT>` modelliert. Sei $r_i := b - Ax_i$ das Residuum der Lösung x_i aus dem i-ten Iterationsschritt. Dann gibt Tabelle 4.5 eine Übersicht über die implemen-

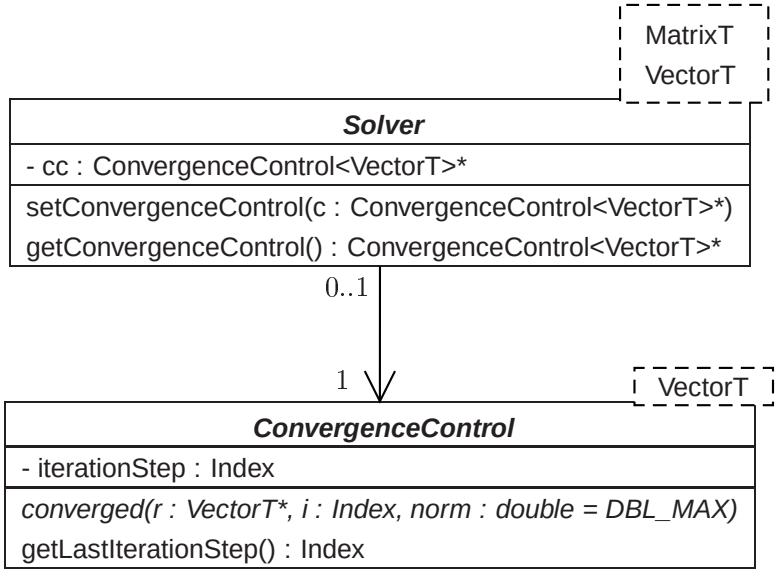


Abbildung 4.8: Löser und Abbruchkontrolle

tierten Abbruchbedingungen. Hierbei sind n und eps Parameter der jeweiligen Klassen, die im Konstruktor angegeben werden. Die Funktion `converged`, die von diesen Klassen implementiert wird, bekommt als Parameter einen Zeiger auf das aktuelle Residuum `*r`, den aktuellen Iterationsschritt `i`, und optional mit dem Gleitkommawert `norm` die jeweilige Norm von r_i, die bei manchen Algorithmen bei der Berechnung gewissermaßen als Nebenprodukt abfällt und dann nicht noch einmal berechnet werden muß.

Abbruchkriterium	**Implementierende Klasse**
$i = n$	`CountControl<VectorT>`
$\|\|r_i\|\|_\infty \leq eps$	`MaximumResidualControl<VectorT>`
$\|\|r_i\|\|_2 \leq eps$	`EuclidianResidualControl<VectorT>`
$\|\|r_i\|\|_2 \,/\, \|\|r_0\|\|_2 \leq eps$	`RelativeEuclidianResidualControl<VectorT>`

Tabelle 4.5: Abbruchkriterien

4.4 Polymorphie

Unter dem Begriff *Polymorphie* versteht man bekanntlich das unterschiedliche Verhalten von Objekten verschiedener Klassen mit derselben Oberklasse bezüglich der namentlich gleichen Operation. Genauer gesagt wird eine solche Funktion, deren Signatur aus der gemeinsamen Oberklasse geerbt wurde, mit einer klassenspezifischen Implementation in

der jeweiligen Unterklasse überschrieben. Dies kommt bei den Matrixklassen natürlich überall dort zum Einsatz, wo in der Klasse `Matrix<T>` lediglich rein virtuelle Funktionen vorhanden sind, also z.B. bei der Funktion `insertItem`.

Dieser Mechanismus bewirkt also, daß zur Laufzeit immer die „richtige“ Funktion für ein Objekt ausgewählt wird. Dies ist deshalb so interessant, weil sich dadurch auf einfache und elegante Weise komplexe Algorithmen schreiben lassen, die verschiedenartige Objekte, ohne Fallunterscheidungen machen zu müssen, in passender Weise behandeln. Ein Beispiel hierfür ist die Matrix-Vektor-Multiplikation, wie sich an folgender Blockmatrix illustrieren läßt :

$$A = \begin{pmatrix} a_{11} & a_{12} & & & & & a_{17} & a_{18} \\ a_{21} & a_{22} & & & & & a_{27} & a_{28} \\ & & a_{33} & a_{34} & & & a_{37} & a_{38} \\ & & a_{43} & a_{44} & & & a_{47} & a_{48} \\ & & & & a_{55} & a_{56} & a_{57} & a_{58} \\ & & & & a_{65} & a_{66} & a_{67} & a_{68} \\ a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & a_{77} & a_{78} \\ a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & a_{87} & a_{88} \end{pmatrix} = \begin{pmatrix} \boxed{A_{11}} & & \boxed{A_{14}} \\ & \boxed{A_{22}} & \boxed{A_{24}} \\ & \boxed{A_{33}} & \boxed{A_{34}} \\ \boxed{A_{41}} & \boxed{A_{42}} & \boxed{A_{43}} \quad \boxed{A_{44}} \end{pmatrix}$$

Hierbei werden die Matrizen $A_{11}, A_{14}, \ldots$ durch Objekte vom Typ

- `ButterflyDenseMatrix<double>`

mit der Dimension 2×2 repräsentiert, während es sich bei A um ein Objekt vom Typ

- `ButterflySparseMatrix<ButterflyDenseMatrix<double>*>`

mit der Dimension 4×4 handelt. Um nun die Matrix A mit einem entsprechend gestalteten Vektor x zu multiplizieren, wird die Template-Funktion `mul` aufgerufen, die im namespace `matrixalgorithms` definiert ist. Sie bekommt als Argument einen Zeiger auf A und auf x und einen Zeiger auf einen passenden Ergebnisvektor, etwa v. `mul` tut nun nichts weiteres, als die entsprechende Methode `mul` von A aufzurufen. Damit ist sichergestellt, daß A in der korrekten Weise mit x multipliziert wird, also etwa nur die Nichtnullblöcke verwendet werden.

Nun müssen aber die Teilblöcke von A ihrerseits wieder mit den entsprechenden Teilblöcken von x multipliziert und die Teilergebnisse aufsummiert werden. Hierfür sind die Funktionen `mul` bzw. `addmul` zuständig, wobei letztere eine Multiplikation und eine Addition durchführt. Ihnen werden als Argument die Zeiger auf die entsprechenden Teilmatrizen und Teilvektoren übergeben. Auch diese Template-Funktionen rufen die entsprechende Methode auf, diesmal aus der Klasse `ButterflyDenseMatrix<double>`, die ja der Typ der Teilmatrizen von A ist.

Dort werden nun die entsprechenden Additions- und Multiplikationsroutinen aus dem namespace `matrixalgorithms` für skalare Einträge aufgerufen, die ebenfalls den Namen `mul` bzw. `addmul` tragen, da sie die überladenen Versionen der oben beschriebenen

Template-Funktionen sind. Die Polymorphie besteht nun darin, daß dreimal der Funktionsname `mul` auftaucht, aber jedesmal mit einer anderen Bedeutung, abhängig vom Typ der Argumente. Abbildung 4.9 zeigt die numerischen Algorithmen, die von den Unterklassen implementiert werden (siehe auch Abschnitt 4.6).

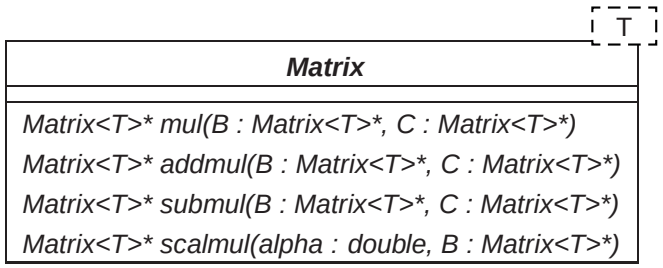


Abbildung 4.9: Überladbare mathematische Operationen der Klasse `Matrix<T>`

Bemerkung 4.5. *An diesem Beispiel wird auch deutlich, wie die zweite Anforderung an die Datenstrukturen aus Abschnitt 4.1.2 in dieser Klassenbibliothek realisiert wurde. Will man eine Matrix deklarieren, die als Einträge wiederum Matrizen hat, so bekommt der umgebende Matrixtyp als Templateparameter einen Zeiger auf den Typ der Matrixeinträge - nicht den Typ der Einträge selbst. Also etwa*

- `ButterflySparseMatrix<Matrix<double>*>* A;`

und nicht

- `ButterflySparseMatrix<Matrix<double>>* A;`

denn nur mit Zeigern läßt sich Laufzeitpolymorphismus in C++ erreichen, nicht etwa mit Referenzen. Die zweite Variante wäre auch kein korrekter C++-Code, da `Matrix<T>` *abstrakt ist und somit nicht instanziiert werden kann. Ein Zeiger auf eine abstrakte Klasse läßt sich jedoch immer deklarieren. So kann* `*A` *als Einträge verschiedene* Typen *von Matrizen beherbergen, die erst zur Laufzeit bekannt sind.*

4.5 Iteratoren

Ein wichtiges Konzept des objektorientierten Designs ist das des Datenzugriffs in Containern über *Iteratoren*. Viele neuere Bibliotheken sind auf diese Weise aufgebaut, die *Standard Template Library* von C++ ([Str98], Teil III) beispielsweise, oder auch die *Matrix Template Library* (*MTL*, siehe [Sie99]).

Algorithmen operieren in der Regel mit bestimmten Daten, die ihrerseits in gewissen Datenstrukturen abgelegt sind. Oft muß also der Container, der die Daten enthält, durchmustert werden. Dies geschieht in objektorientierten Bibliotheken mit sogenannten *Iteratoren*. Hierbei handelt es sich um eine weitere Abstraktionsschicht, die zwischen den Algorithmen und den Datenstrukturen eingezogen wird. Sie ermöglichen den effizienten, sukzessiven Zugriff auf Elemente eines Containers in einer gewissen Reihenfolge, ohne allzuviel über dessen Implementation preiszugeben. Ein Algorithmus braucht also nicht mehr

über die spezielle Implementation eines Containers Bescheid zu wissen, sondern kann einmal allgemein formuliert werden und funktioniert dann mit einer Vielzahl von Containern (im Idealfall), die lediglich ihren eigenen Iterator bereitstellen müssen.

Abbildung 4.10: Iteratoren als Mittler zwischen Daten und Algorithmen

Container und ihre Iteratoren hängen enger zusammen als Algorithmen und Iteratoren, weshalb in Abbildung 4.10 die Abhängigkeit zwischen diesen auch in beide Richtungen zeigt. Ein Container muß seinen Iterator kennen, da er ihn beispielsweise am Beginn einer Iteration erzeugen muß. Andererseits muß ein Iterator direkten Zugriff auf die Daten des Containers haben - meistens ist der Iterator eine `friend`-Klasse des Containers. Hingegen muß ein Algorithmus zwar wissen, wie er einen Iterator zu benutzen hat, jedoch weiß der Iterator nichts über irgendeinen Algorithmus.

Die Besonderheit bei Matrizen liegt nun in der zweidimensionalen Anordnung der Daten. Will man alle Matrixeinträge mit einem Iterator durchlaufen, so muß man sich für eine bestimmte Durchlaufrichtung entscheiden (etwa zeilenweise). Eine andere Lösung besteht darin, zwei Stufen von Iteratoren zu verwenden : Eine Art von Iteratoren, mit denen man über die Zeilen, und eine andere, mit denen man durch die Spalten läuft. Die letztere Form ist unter anderem in der MTL ([Sie99], Kapitel 2) verwirklicht.

In **lins** sind beide Iteratorkonzepte implementiert worden, da für unterschiedliche Algorithmen mal das eine, mal das andere Konzept effizienter ist. Beispielsweise ist für die Matrix-Vektor-Multiplikation das lineare Durchlaufen schneller (vgl. Abschnitt 4.6.1), während sich das Verfahren des Rückwärtseinsetzens besser mit den zweistufigen Iteratoren realisieren läßt. Zudem verfügen nur die Unterklassen von `RowWiseMatrix<T>` über eigene Iteratoren, da sie nur hier benötigt werden.

Die Iteratoren sind in dieser Bibliothek als eigene Klassen verwirklicht. Für das zweistufige Iteratorkonzept sind dies zwei Typen für jede Matrixklasse, deren Namen innerhalb des Namensbereichs der Matrixklasse durch `FirstLevelIterator` bzw. `SecondLevelIterator` gegeben ist. Die qualifizierten Namen wären also beispielsweise für die Klasse `TriangularSparseMatrix<T>` :

- `TriangularSparseMatrix<T>::FirstLevelIterator`, bzw.
- `TriangularSparseMatrix<T>::SecondLevelIterator`.

Darüber hinaus gibt es einen weiteren Iteratortyp, der zum linearen Durchmustern dient. Sein Name innerhalb einer Klasse ist `LinearIterator`, d.h. sein qualifizierter Name lautet für die Klasse `TriangularSparseMatrix<T>` :

- `TriangularSparseMatrix<T>::LinearIterator`.

Jede zu `RowWiseMatrix<T>` konforme Matrixklasse hat nun die folgenden Funktionen :

- `begin()` : liefert einen `FirstLevelIterator`, der auf die erste Zeile, bzw. bei transponierten Matrizen die erste Spalte zeigt.
- `end()` : liefert einen `FirstLevelIterator`, der hinter die letzte Zeile, bzw. bei transponierten Matrizen die letzte Spalte zeigt.
- `linearBegin()` : liefert einen `LinearIterator`, der auf das erste Element zeigt.
- `linearEnd()` : liefert einen `LinearIterator`, der hinter das letzte Element zeigt.

Ein `FirstLevelIterator` hat nun wiederum die folgenden Funktionen :

- `begin()` : liefert einen `SecondLevelIterator`, der auf das erste Element der Zeile (bzw. Spalte bei transponierten Matrizen) zeigt, auf die der `FirstLevelIterator` selbst zeigt.
- `end()` : liefert einen `SecondLevelIterator`, der hinter das letzte Element der Zeile (Spalte bei transponierten Matrizen) zeigt, auf die der `FirstLevelIterator` selbst zeigt.
- `position()` : liefert den aktuellen Index der Zeile (Spalte bei transponierten Matrizen), auf die der `FirstLevelIterator` selbst zeigt.
- `operator++()` : setzt den `FirstLevelIterator` um eine Zeile (bzw. Spalte) weiter.

Ein `SecondLevelIterator` dient zum Durchlaufen der eigentlichen Elemente. Er bietet die folgende Schnittstelle :

- `row()` : liefert den aktuellen Index der Zeile, auf die der `SecondLevelIterator` zeigt.
- `column()` : liefert den aktuellen Index der Spalte, auf die der `SecondLevelIterator` zeigt.
- `operator++()` : setzt den `SecondLevelIterator` auf das nächste Nichtnullelemet der aktuellen Zeile (bzw. Spalte).
- `operator*()` : liefert eine Referenz auf das aktuelle Element der Matrix, auf das der `SecondLevelIterator` zeigt.

Ein `LinearIterator` hat die gleiche Schnittstelle wie ein `SecondLevelIterator`, er bietet dieselben Funktionssignaturen. Jedoch ist hier im allgemeinen nicht sichergestellt, daß die beim Durchmustern mit `columns()` erzeugte Folge von Indizes monoton wachsend ist, denn es wird ja die ganze Matrix durchlaufen, und nicht nur eine Zeile, wie beim `SecondLevelIterator`.

4.6 Angepaßte Algorithmen

Um die Datenstruktur, die durch die Klasse `ButterflySparseMatrix<T>` realisiert wird, optimal auszunutzen, ist eine Anpassung verschiedener Algorithmen notwendig. Durch die Aufteilung in rechten oberen Teil, Diagonale und linken unteren Teil ist es erforderlich, viele Verfahren anders aufzuteilen.

Funktionen, die einfache Berechnungen mit Vektoren und Matrizen durchführen, sind im namespace `matrixalgorithms` zusammengefaßt. Tabelle 4.6 gibt eine Übersicht über die dort enthaltenen Funktions-Templates.

mathematische Operation	**Funktion**
$z \Leftarrow x + y$	`add(VektorT* x, VektorT* y, VektorT* z)`
$z \Leftarrow x - y$	`sub(VektorT* x, VektorT* y, VektorT* z)`
$z \Leftarrow Ax$	`mul(MatrixT* A, VektorXT* x, VektorZT* z)`
$z \Leftarrow z + Ax$	`addmul(MatrixT* A, VektorXT* x, VektorZT* z)`
$z \Leftarrow z - Ax$	`submul(MatrixT* A, VektorXT* x, VektorZT* z)`
$z \Leftarrow \alpha x$	`scalmul(double alpha, MatrixT* x, MatrixT* z)`
$z \Leftarrow \alpha x + y$	`addscalmul(MatrixT* x, MatrixT* y, MatrixT* z, double alpha)`

Tabelle 4.6: Funktionen im Namespace `matrixalgorithms`

Hierbei ist $A \in \mathbf{R}^{n \times n}$ und `A` ein Zeiger auf das entsprechende Matrixobjekt, $x, y, z \in \mathbf{R}^n$ und `x`,`y` und `z` sind die Zeiger auf die jeweiligen Vektorobjekte. Außerdem ist $\alpha \in \mathbf{R}$ und `alpha` der entsprechende Skalar. Sämtliche Funktionen gehen davon aus, daß der Speicher für den Vektor `*z` bereitgestellt und dieser so dimensioniert ist, daß er das Resultat der Berechnung aufnehmen kann. Am Ende der Operation zeigt `z` also auf das Ergebnis. Die Typen `VektorT`, `MatrixT`, `VektorXT` und `VektorZT` sind die Template-Argumente der Funktionen.

4.6.1 Matrix-Vektor-Multiplikation

Eine sehr wichtige und zeitkritische Operation bei iterativen Verfahren ist die Berechnung eines Produktes $Ax = y$, wobei $x, y \in \mathbf{R}^n$ Vektoren sind und $A \in \mathbf{R}^{n \times n}$ eine entsprechende Matrix. Fast noch wichtiger ist die kombinierte Operation $Ax + z = z$, mit $z \in \mathbf{R}^n$, die durch `addmul` implementiert ist, denn sie wird in den meisten Algorithmen häufiger als `mul` verwendet.

Unter Berücksichtigung des mit Formel 2.15 eingeführten Splittings $A = L + D + R$ läßt sich nun aus diesen Funktionen die Multiplikation für eine zu `ButterflyMatrix<T>`

konforme Klasse zusammensetzen, wie in Algorithmus 4.1 zu sehen ist.

Algorithmus 4.1 `addmul`

$z \Leftarrow z + Dx$
$z \Leftarrow z + Lx$
$z \Leftarrow z + Rx$

Die Multiplikation für `Butterfly`-Matrizen ist also zurückgeführt auf die Multiplikation von Dreiecks- und Diagonalmatrizen.

Schaut man sich nun den entsprechenden Algorithmus für die Matrizen der Klasse `TriangularSparseMatrix<T>` an, so gibt es mit den in Abschnitt 4.5 vorgestellten Iteratoren prinzipiell zwei mögliche Arten seiner Implementierung. Die eine besteht darin, zweistufige Iteratoren zu benutzen, woraus sich die mehr oder weniger klassische Matrix-Vektor-Multiplikation mit zwei Schleifen ergibt (vgl. Alg. 4.2).

Algorithmus 4.2 `matrixalgorithms::addmul` mit zweidimensionalen Iteratoren

```
VektorZT* matrixalgorithms::addmul(MatrixT* A, VektorXT* x, VektorZT* z)
{
  typename MatrixT::MatrixFirstLevelIterator rit = A->begin();
  typename MatrixT::MatrixFirstLevelIterator ritend = A->end();
  typename MatrixT::MatrixSecondLevelIterator cit, citend;
  for(; rit != ritend, rit++)
  {
    cit = rit.begin();
    citend = rit.end();
    for(; cit != citend; cit++)
      matrixalgorithms::addmul(*cit, (*x)[cit.column()], (*z)[cit.row()]);
  }
  return C;
}
```

Die andere Möglichkeit besteht in der Verwendung von eindimensionalen Iteratoren, wie in Alg. 4.3 gezeigt. Diese Version kommt mit nur einer Schleife aus und besitzt nicht nur den Vorteil einer größeren Übersichtlichkeit des Programmcodes, sondern ist überdies auch noch schneller, wie in Tabelle 4.7 demonstriert wird. Hierbei wurden die Operationen jeweils 10000mal hintereinander für verschiedene Matrizen $A_n \in \mathbf{R}^{n \times n}$ ausgeführt. Die mit **adr** erzeugten Matrizen haben 28553 Nichtnulleinträge für $n = 4225$ bzw. 114441 für $n = 16641$. Zur verwendeten Testhardware vgl. Anhang C.

Algorithmus 4.3 `matrixalgorithms::addmul` mit eindimensionalen Iteratoren

```
VektorZT* matrixalgorithms::addmul(MatrixT* A, VektorXT* x, VektorZT* z)
{
  typename MatrixT::MatrixLinearIterator lit = A->linearBegin();
  typename MatrixT::MatrixLinearIterator litend = A->linearEnd();
  for(; lit != litend; lit++)
    matrixalgorithms::addmul(*lit, (*x)[lit.column()], (*z)[lit.row()]);
  return C;
}
```

Die Laufzeitunterschiede der beiden Implementationen erklären sich hauptsächlich durch den zusätzlichen Overhead der zweiten Schleife in Algorithmus 4.2, die sich innerhalb der ersten befindet. Hier muß die Pipeline der CPU immer neu angesetzt werden, was hingegen bei Algorithmus 4.3 vermieden wird.

Matrix	**Hardware**	`addmul` **mit zweidimensionalen Iteratoren**	`addmul` **mit linearen Iteratoren**
A_{4225}	Compaq Alpha	26.8 sec	16.9 sec
A_{4225}	Intel PIII	51.1 sec	40.3 sec
A_{16641}	Compaq Alpha	117.9 sec	76.1 sec
A_{16641}	Intel PIII	256.9 sec	215.3 sec

Tabelle 4.7: Laufzeitvergleich der verschiedenen Implementierungen von `addmul`

Eine weitere Beschleunigung kann man im Spezialfall symmetrischer Matrizen erreichen. In diesem Fall sind die Daten für obere und untere Dreiecksmatrix identisch, sie werden, wie bereits beschrieben, auch nur einmal abgespeichert und lediglich unterschiedlich adressiert. Wie in [Str98], Abschnitt 22.4.7., Seite 724, erwähnt, empfiehlt es sich, mehrfache Schleifen über dieselben Daten zu vermeiden, was zu der Version von `addmul` führt, die in Algorithmus 4.4 zu sehen ist.

Algorithmus 4.4 `matrixalgorithms::symmetricaddmul`

```
VektorZT* matrixalgorithms::addmul(MatrixT* A, VektorXT* x, VektorZT* z)
{
  typename MatrixT::MatrixLinearIterator lit = A->linearBegin();
  typename MatrixT::MatrixLinearIterator litend = A->linearEnd();
  for(; lit != litend; lit++)
  {
    matrixalgorithms::addmul(*lit, (*x)[lit.column()], (*z)[lit.row()]);
    matrixalgorithms::addmul(*lit, (*x)[lit.row()], (*z)[lit.column()]);
  }
  return C;
}
```

Die Tabelle 4.8 zeigt das Laufzeitverhalten dieser Funktion. Es wurden hierbei die symmetrischen Anteile $A_n^{sym} = \frac{1}{2}(A_n + A_n^T)$ der Matrizen aus Tabelle 4.7 verwendet. Im Vergleich zur normalen `addmul`-Funktion ergeben sich also Laufzeitreduktionen um bis zu 21 %.

Matrix	**Hardware**	`symmetricaddmul`
A_{4225}^{sym}	Compaq Alpha	13.3 sec
A_{4225}^{sym}	Intel PIII	32.4 sec
A_{16641}^{sym}	Compaq Alpha	66.8 sec
A_{16641}^{sym}	Intel PIII	170.0 sec

Tabelle 4.8: Laufzeiten von `symmetricaddmul`

4.6.2 ILU-Zerlegung

Um die ILU-Zerlegung in der Sparse-Skyline-Variante, wie sie von der Klasse `Butterfly-SparseMatrix<T>` verwendet wird, effizient durchführen zu können, muß Algorithmus 2.15 entsprechend angepaßt werden.

Die Änderungen ergeben sich in erster Linie dadurch, daß man bei dem Verfahren die Durchlaufrichtung der Matrix berücksichtigen muß, um keine Zeit mit dem Suchen bestimmter Elemente zu verschwenden. Die Kunst besteht also in der richtigen Anordnung der Schleifen. Algorithmus 4.5 zeigt diese Variante.

Algorithmus 4.5 ILU-Variante für zu `ButterflyMatrix<T>` konforme Klassen

```
1:  Setze U := (u_ij)_{i,j=1}^n mit u_ij = a_ij für j > i, 0 sonst
2:  Setze D := (d_ij)_{i,j=1}^n mit d_ij = a_ij für j = i, 0 sonst
3:  Setze L := (l_ij)_{i,j=1}^n mit l_ij = a_ij für i > j, 0 sonst
4:  for i = 1,...,n do
5:    for m = 1,...,i-1 do
6:      if m ∈ M_S^A(i) then
7:        if m ∈ M_Z^A(i) then
8:          d_ii ⇐ d_ii - l_im u_mi
9:        end if
10:       for k = i+1,...,n do
11:         if k ∈ M_S^A(i) ∧ m ∈ M_Z^A(k) then
12:           l_ki ⇐ l_ki - l_km u_mi
13:         end if
14:       end for
15:     end if
16:   end for
17:   for m = 1,...,i-1 do
18:     if m ∈ M_Z^A(i) then
19:       for k = i+1,...,n do
20:         if k ∈ M_Z^A(i) ∧ m ∈ M_S^A(k) then
21:           u_ik ⇐ u_ik - l_im u_mk
22:         end if
23:       end for
24:     end if
25:   end for
26:   for k = i+1,...,n do
27:     if k ∈ M_Z^A(i) then
28:       u_ik ⇐ u_ik / d_ii
29:     end if
30:   end for
31: end for
```

Im ersten Teil von Zeile 5 - 14, in der L berechnet wird, wird nun in der innersten Schleife über k durch die i. und die m. Spalte von A bzw. L iteriert. Dies geschieht in der Praxis durch einen `SecondLevelIterator`, der eigentlich über die Zeile von L^T läuft. Entsprechend wird U im zweiten Teil von Zeile 15 - 29 berechnet. Dort wird in der innersten Schleife nun über k durch die i. und die m. Zeile von U iteriert, was in der Klassenbibliothek mit einem `SecondLevelIterator` implementiert ist.

4.6.3 IC-Zerlegung

Aus den gleichen Gründen wie zuvor im Abschnitt über die ILU-Zerlegung beschrieben, muß auch die Incomplete-Cholesky-Zerlegung an die verwendete Datenstruktur angepaßt werden. Auch dieses Verfahren wird mit zweidimensionalen Iteratoren implementiert. Dadurch werden die Schleifen, die in einer weiteren if-Abfrage überprüfen, ob ihr Laufindex

in einem gewissen Zeilen- bzw. Spaltenmuster der Matrix enthalten ist (wie in den Zeilen 5, 8 und 16 in Algorithmus 4.6), auf elegante und effiziente Weise realisiert.

Algorithmus 4.6 IC-Variante für zu `ButterflyMatrix<T>` konforme Klassen

```
 1: Setze D := (d_ij)_{i,j=1}^n mit d_ij = a_ij für j = i, 0 sonst
 2: Setze L := (l_ij)_{i,j=1}^n mit l_ij = a_ij für i > j, 0 sonst
 3: for k = 1, ..., n do
 4:    for j = 1, ..., k − 1 do
 5:       if j ∈ M_Z^A(k) then
 6:          d_kk ⇐ d_kk − l_kj^2
 7:          for i = k + 1, ..., n do
 8:             if j ∈ M_Z^A(i) ∧ i ∈ M_Z^A(k) then
 9:                l_ik ⇐ l_ik − l_ij l_kj
10:             end if
11:          end for
12:       end if
13:    end for
14:    d_kk ⇐ √d_kk
15:    for i = k + 1, ..., n do
16:       if i ∈ M_Z^A(k) then
17:          l_ik ⇐ l_ik / d_kk
18:       end if
19:    end for
20: end for
```

Kapitel 5

Numerische Experimente

In diesem Kapitel soll ein Vergleich zwischen den verschiedenen Vorkonditionierern für das GMRES-Verfahren gezogen werden. Anhand eines Modellproblems werden hierbei die Laufzeiten von GMRES(m) für unterschiedliche Gittergrößen, Diffusionskoeffizienten ε und Restartlängen m untersucht.

5.1 Beschreibung des Modellproblems

Als numerischer Benchmark wurde das bereits in [Pri96] untersuchte Problem der Rotationsströmung verwendet. Hierbei handelt es sich um ein zweidimensionales, stationäres Konvektions-Diffusions-Problem der Form

$$-\varepsilon\Delta u + \vec{b}\cdot\nabla u = f \quad \text{in } \Omega. \tag{5.1}$$

Dies ist ein Spezialfall von (1.20) mit $c \equiv 0$. Für das Gebiet Ω wurde das Einheitsquadrat $\Omega = (0,1)\times(0,1)$ verwendet. Zur Rotationsströmung wird dieses Problem durch das Geschwindigkeitsfeld

$$\vec{b}(x,y) := \begin{pmatrix} (2y-1)(1-(2x-1)^2) \\ 4y(2x-1)(y-1) \end{pmatrix}.$$

Ferner seien auf dem Rand die folgenden Dirichlet-Bedingungen vorgegeben :

$$\begin{aligned} u &= -0.5 \quad \text{auf} \quad \{(0\ y)^T : 0 \le y \le 1\} \\ u &= \ \ 0.5 \quad \text{auf} \quad \{(1\ y)^T : 0 \le y \le 1\} \\ u &= \ \ 0 \quad \text{sonst.} \end{aligned}$$

Außerdem wird stets $f \equiv 0$ gesetzt. Sämtliche Testfälle wurden mit strukturierten Gittern und unterschiedlichen Feinheitsgraden h gerechnet. Hierbei wurde das Einheitsquadrat in kongruente rechtwinklige (und gleichschenklige) Dreiecke der Höhe $h \in \{\frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}\}$ unterteilt. Weiterhin wurde durchgehend die SUPG-Stabilisierung (SDFEM, vgl. Abschnitt 1.3.2) benutzt.

In den folgenden vier Abschnitten werden die diskreten Lösungen (auf dem feinsten Gitter) für $\varepsilon = 1, 10^{-2}, 10^{-4}, 10^{-6}$ dargestellt. Außerdem werden jeweils die Zeiten für die verschiedenen Lösungsverfahren bzw. Vorkonditionierer auf den vier verschiedenen

Gittern verglichen. Als Abbruchkriterium diente die Reduktion der euklidischen Norm des (nichtvorkonditionierten) Residuums um den Faktor 10^{-6}, also $||r_i||_2 \,/\, ||r_0||_2 < 10^{-6}$.

In den jeweils vier Balkendiagrammen sind die Löserzeiten für GMRES(m) nach oben abgetragen. Ein schwarzer Teilbalken gibt hierbei die benötigte Zeit für die Initialisierung des Lösers an, insbesondere also auch die Konstruktion des Vorkonditionierers. Weiterhin stehen die vier Gruppen von Balken für die Restartlängen $m = 8, 12, 16, 20$. Die jeweils fünf direkt nebeneinanderstehenden Balken geben dabei die Löserzeiten für den GMRES(m)

- ohne Vorkonditionierung,
- mit JOR-Vorkonditionierung,
- mit SOR-Vorkonditionierung,
- mit SSOR-Vorkonditionierung,
- mit ILU-Vorkonditionierung

an (von links nach rechts). In den Tabellen wurden zusätzlich die Anzahl der Iterationen (Restarts) angegeben, die das Verfahren bis zum Erreichen des Konvergenzkriteriums benötigte. Trat nach einer bestimmten Anzahl von Restarts keine Konvergenz ein, so wurde dies in der Tabelle mit einem Strich deutlich gemacht.

Als Relaxationsparameter wurde jeweils der als optimal ermittelte verwendet (vgl. Anhang B). Es wurde die Startlösung x_0 für jedes Verfahren so gewählt, daß alle Vektorkomponenten den gleichen Wert haben und zusätzlich $||x_0||_2 = 1$ gilt. Für die Matrizen ergeben sich folgende Eckdaten :

h	n	nnz
1/32	1089	7113
1/64	4225	28553
1/128	16641	114441
1/256	66049	458249

Tabelle 5.1: Verhältnis von Matrixdimension und Nichtnulleinträgen

Hierbei ist n die Dimension (Kantenlänge) der Matrix und nnz die Anzahl der Nichtnulleinträge. Variiert man $\varepsilon > 0$, so ändern sich lediglich die Werte der Einträge, nicht aber das Besetzungmuster der Matrix, so daß die Größe nnz für ein Gitter konstant bleibt.

Vergleicht man die Ergebnisse, so läßt sich feststellen, daß eine Vorkonditionierung zumindest meist eine Reduktion der Iterationen (Restarts) zur Folge hat. Dieses Verhalten ist umso drastischer, je „einfacher“ das Problem ist (bei $\varepsilon = 1, 10^{-2}$). Dort tritt auch eine z.T. erhebliche Verminderung der Laufzeit ein. Aber bereits bei $\varepsilon = 10^{-2}$ läßt sich beim JOR-Vorkonditionierer erkennen, daß die Reduktion der Iterationszahlen nicht immer ausreicht, um auch die absolute Laufzeit unter die des nichtvorkonditionierten GMRES zu drücken.

Da ein Vorkonditionierer in jedem Iterationsschritt zusätzliche Operationen erfordert, muß die Konditionsverbesserung schon beträchtlich sein, da sonst die eingesparten Iterationen wieder durch die zusätzlichen Kosten zunichte gemacht werden. Ein SSOR-Schritt beispielsweise ist bezüglich des Rechenaufwandes in etwa genauso „teuer“ wie eine Matrixmultiplikation. Vernachlässigt man die reinen Vektoroperationen, so ist ein entsprechend vorkonditionierter GMRES-Schritt nun doppelt so aufwendig wie ein nicht vorkonditionierter. Ergibt sich mit diesem Verfahren nur eine Reduktion der Restarts um die Hälfte, so handelt es sich hierbei um ein Nullsummenspiel, denn die Laufzeit bleibt nahezu unverändert. Aus diesem Grund bringen Vorkonditionierer in vielen Fällen sogar eine Laufzeitverschlechterung.

Letzteres kann man insbesondere bei kleinem ε und feinerem Gitter beobachten. Hier ist lediglich vereinzelt mal der SOR, mal der SSOR und mal der ILU schneller. Auch wird deutlich, daß der SOR oder der SSOR mit gut gewähltem Relaxationsparameter eine ähnlich gute oder teilweise sogar bessere Konvergenzbeschleunigung erbringen als der ILU. Jedoch ist der große Vorteil des ILU seine Parameterunabhängigkeit – die ILU-Zerlegung wird, unabhängig von A, nach einem vorgegebenen Algorithmus konstruiert. Die Bestimmung des optimalen Parameters für ein Relaxationsverfahren und eine beliebige Matrix A ist jedoch schwierig, und oft nur durch entsprechende Experimente (vgl. Anhang B) möglich.

Ebenso wie den zusätzlichen Aufwand durch die Vorkonditionierung muß man auch die Einsparung von Restarts durch eine größere Restartlänge m gegenüber den dadurch entstehenden höheren Kosten bei der Transformation der Hessenbergmatrix abwägen. Eine größere Restartlänge erbringt bei großem ε keine Laufzeitvorteile. Hingegen läßt sich, wie schon in [Pri96] beschrieben, bei kleinerem ε hierdurch eine Stabilisierung und damit eine teilweise deutliche Reduktion der Laufzeit erreichen. Beispielsweise sank die Laufzeit bei $m = 40$, ILU-Vorkonditionierung, $\varepsilon = 10^{-6}$ und $h = 1/32$ auf 1.16 Sekunden (im Vergleich zu 21.87 Sekunden bei $m = 20$), bei $h = 1/64$ auf 10.53 Sekunden (42.75 Sek. bei $m = 20$) und bei $h = 1/128$ auf 79.61 Sekunden (118.195 Sek. bei $m = 20$). Bei $h = 1/256$ sank die Zeit immerhin auf 748.48 Sekunden (852.17 Sek. bei $m = 20$).

Bei der Bewertung der Rechenzeiten darf zudem nicht außer acht gelassen werden, daß die Berechnungen auf einem Multitasking- und Multiusersystem durchgeführt wurden, d.h., daß parallel ablaufende Prozesse die Zeitmessung zumindest geringfügig beeinträchtigen konnten. Zwar zählt die C-Funktion `clock()` nur die vergangenen CPU-Takte, die sich mit dem eigenen Prozess befasst haben, jedoch kann ein weiterer rechenintensiver konkurrierender Prozess auf demselben Rechner dazu führen, daß beispielsweise der Cache laufend die „falschen“ Daten (nämlich die des anderen Prozesses) enthält. Damit wird jedes Caching ad absurdum geführt, was das Rechen natürlich verlangsamt. Jedoch sollte sich dieser Effekt nicht allzusehr auf die gemessenen Zeiten auswirken.

Es wurde abschließend auch noch exemplarisch ein Beispiel mit $h = \frac{1}{512}$ und $\varepsilon = 10^{-2}$ gerechnet. **adr** generierte daraus eine Matrix der Dimension $n = 263169$ mit $nnz = 1833994$ Nichtnullelementen. GMRES(m) mit der Restartlänge $m = 12$ und ILU-Vorkonditionierung löste das Problem in 38 Minuten und benötigte hierfür 318 Restarts.

5.2 Rotationsströmung, $\varepsilon = 1$

5.2.1 Plot der diskreten Lösung

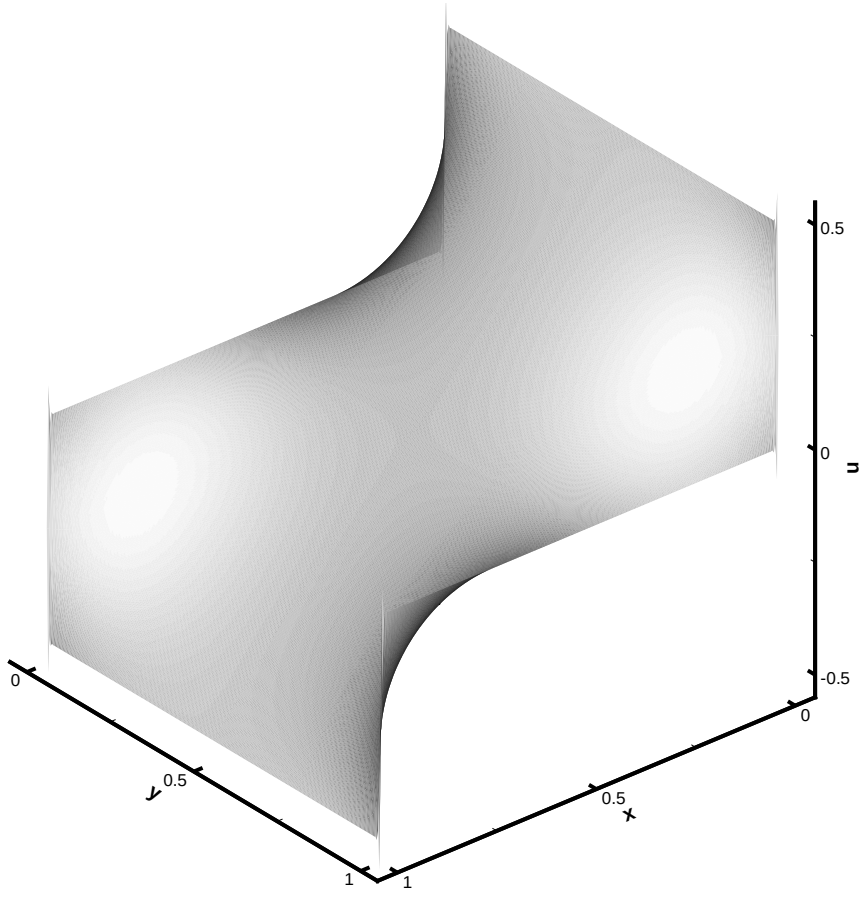
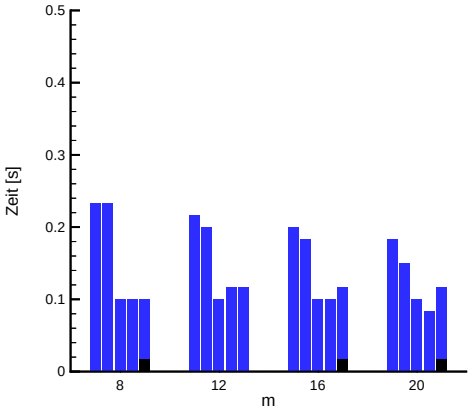


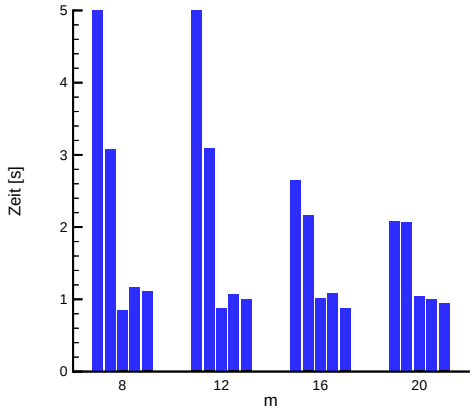
Abbildung 5.1: Rotationsströmung, $\varepsilon = 1$, $h = \frac{1}{256}$

5.2.2 Laufzeitvergleich



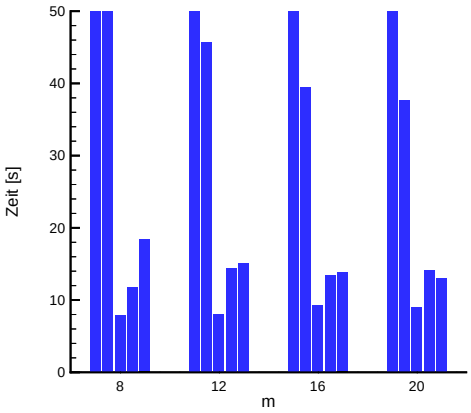
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	48	40	12	10	9
12	26	21	7	7	6
16	16	13	5	4	4
20	12	8	4	3	3

$n = 33 \times 33$



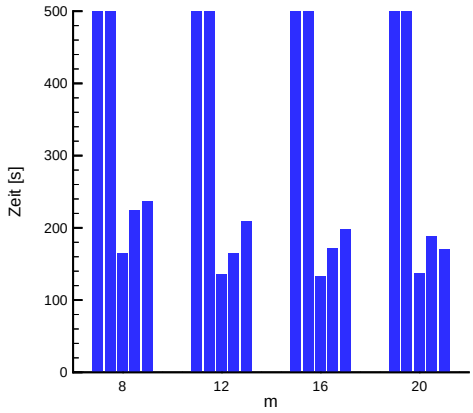
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	133	25	26	26
12	-	66	17	15	14
16	51	39	11	10	9
20	29	27	9	7	7

$n = 65 \times 65$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	450	48	58	75
12	-	220	34	40	39
16	-	129	25	27	25
20	-	86	19	21	18

$n = 129 \times 129$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	1392	195	197	208
12	-	756	100	92	120
16	-	446	68	69	78
20	-	289	54	56	52

$n = 257 \times 257$

5.3 Rotationsströmung, $\varepsilon = 10^{-2}$

5.3.1 Plot der diskreten Lösung

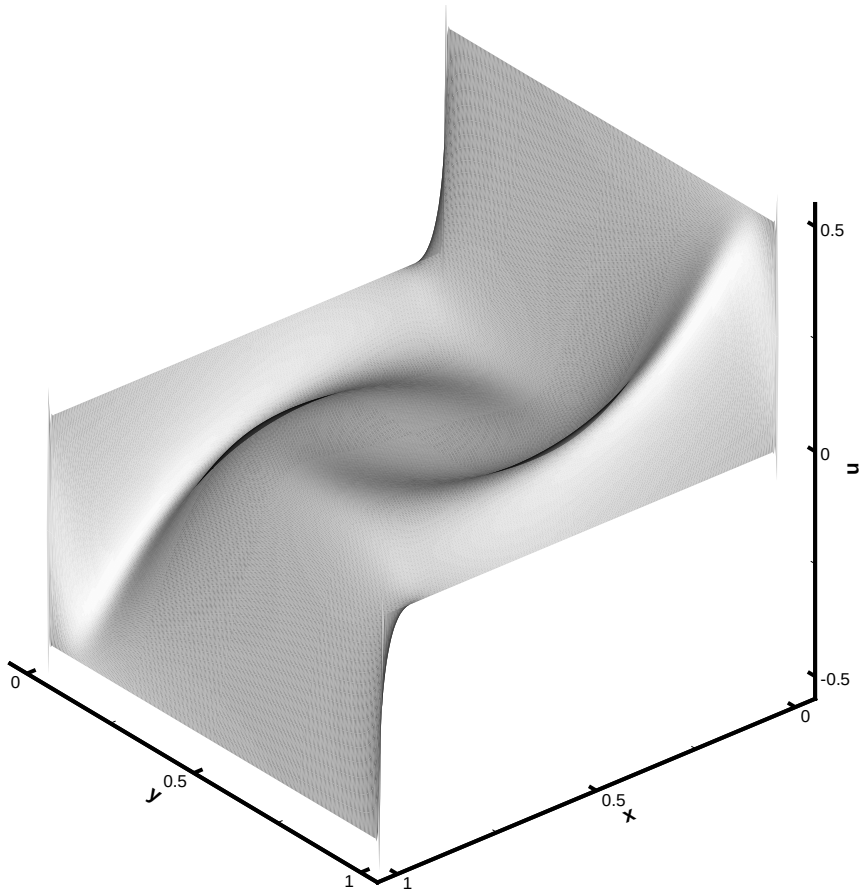
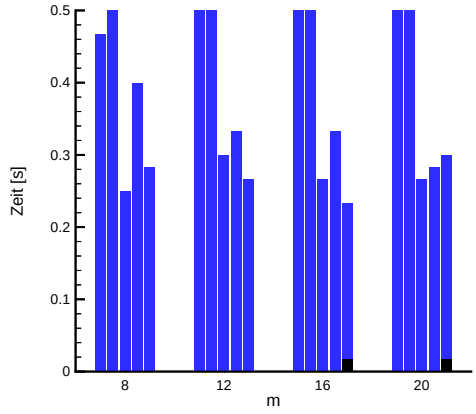


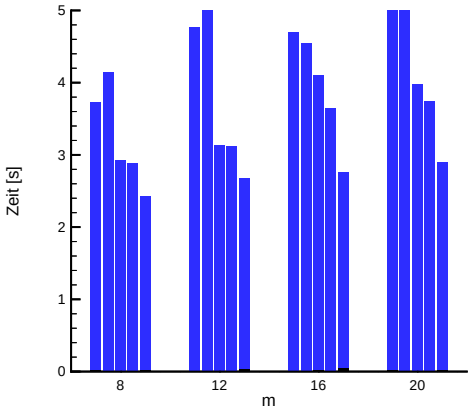
Abbildung 5.2: Rotationsströmung, $\varepsilon = 10^{-2}$, $h = \frac{1}{256}$

5.3.2 Laufzeitvergleich



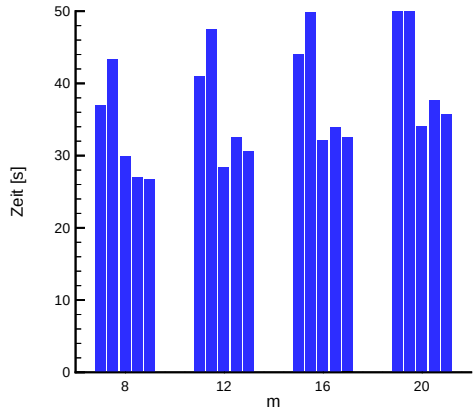
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	99	84	33	29	26
12	65	55	21	17	15
16	49	41	14	11	10
20	39	33	11	8	8

$n = 33 \times 33$



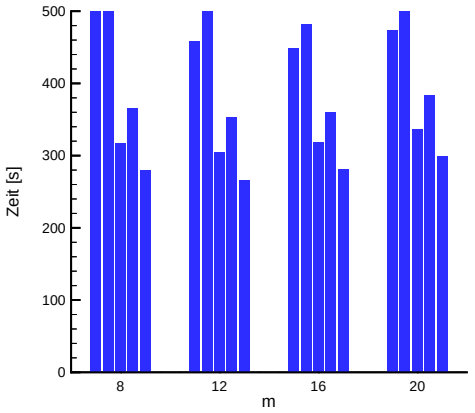
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	194	179	88	66	58
12	124	114	59	45	39
16	90	83	45	33	29
20	73	67	35	26	23

$n = 65 \times 65$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	390	379	183	138	119
12	232	225	117	90	79
16	167	163	87	69	60
20	131	128	71	56	49

$n = 129 \times 129$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	945	940	371	276	240
12	487	482	224	173	152
16	323	321	163	128	113
20	244	242	129	102	91

$n = 257 \times 257$

5.4 Rotationsströmung, $\varepsilon = 10^{-4}$

5.4.1 Plot der diskreten Lösung

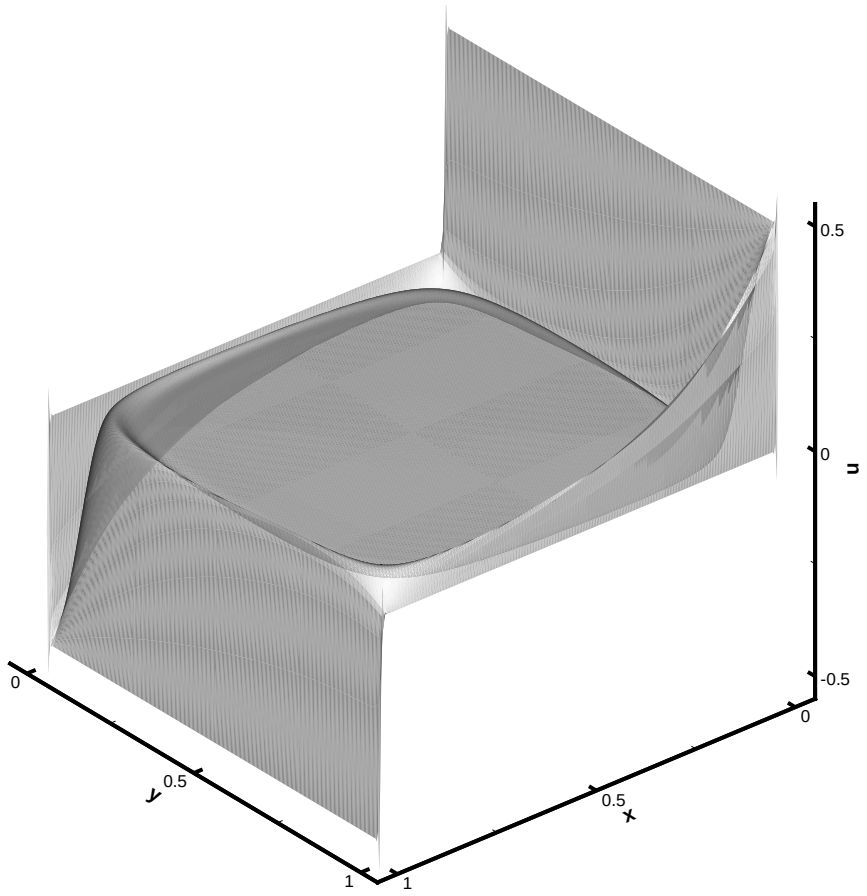
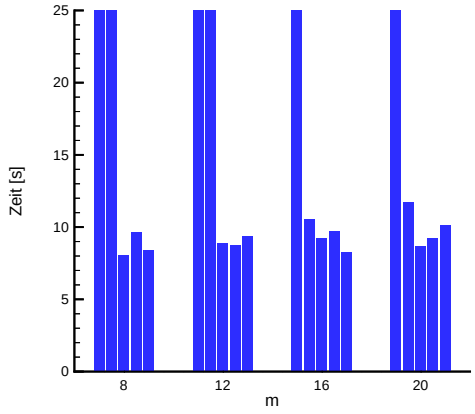


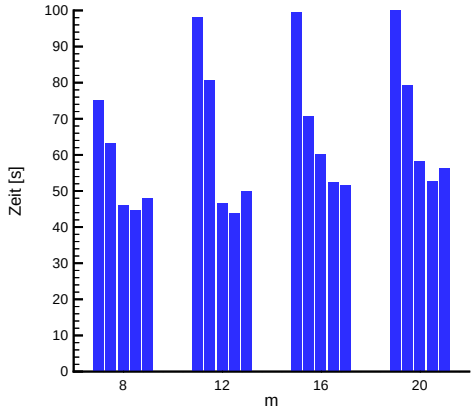
Abbildung 5.3: Rotationsströmung, $\varepsilon = 10^{-4}$, $h = \frac{1}{256}$

5.4.2 Laufzeitvergleich



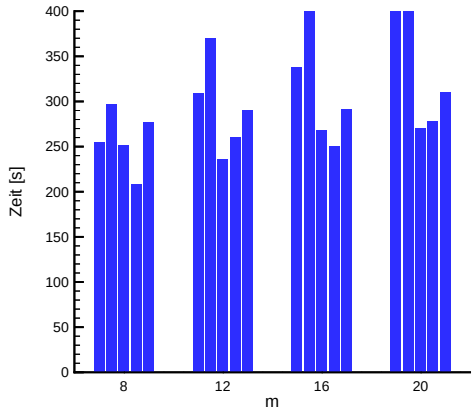
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	-	1047	709	798
12	-	-	637	450	528
16	-	813	477	330	376
20	-	657	373	263	288

$n = 33 \times 33$



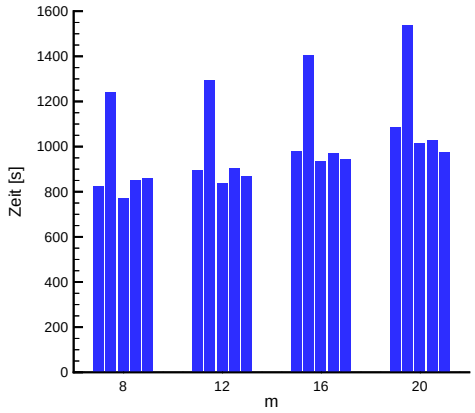
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	-	1396	1004	1106
12	-	1759	894	631	719
16	1926	1316	664	483	544
20	1528	1052	524	364	432

$n = 65 \times 65$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	-	1548	1084	1173
12	1834	1794	1026	713	767
16	1376	1372	740	516	553
20	1097	1089	591	407	437

$n = 129 \times 129$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	1469	-	935	692	744
12	953	1283	633	476	501
16	722	944	489	350	372
20	565	750	396	284	295

$n = 257 \times 257$

5.5 Rotationsströmung, $\varepsilon = 10^{-6}$

5.5.1 Plot der diskreten Lösung

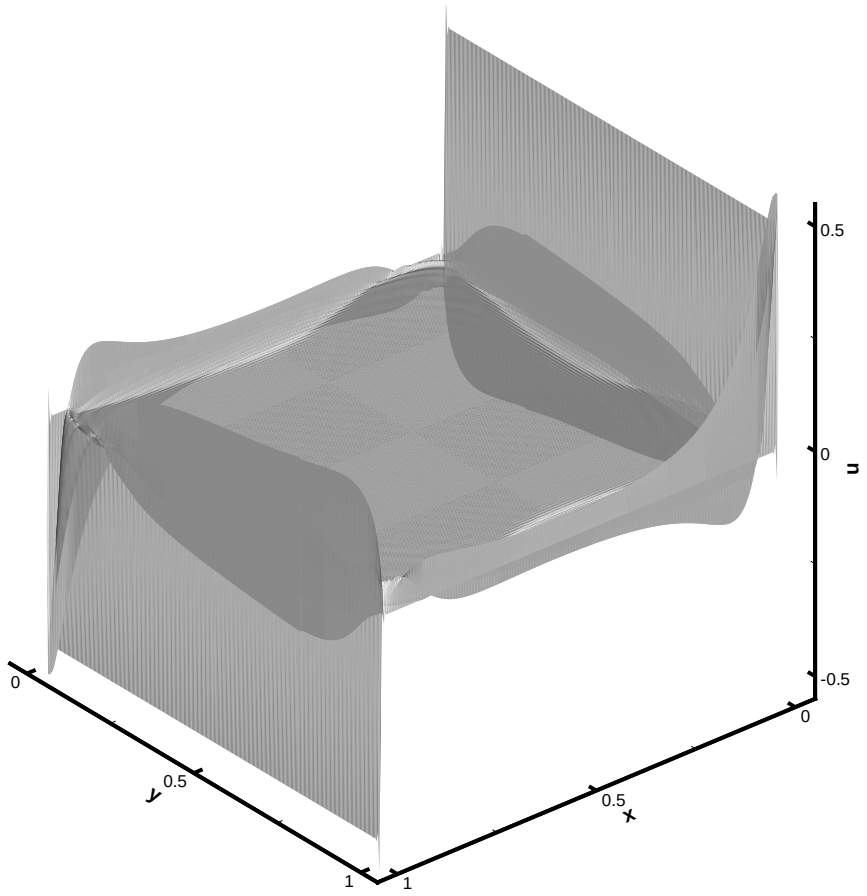
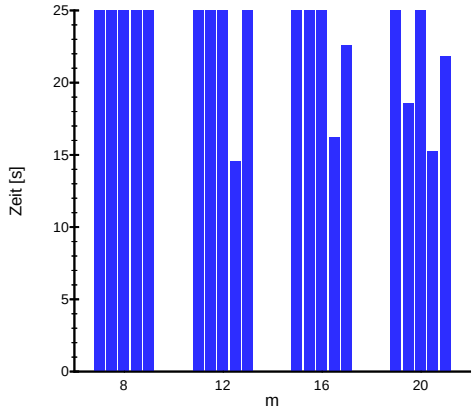


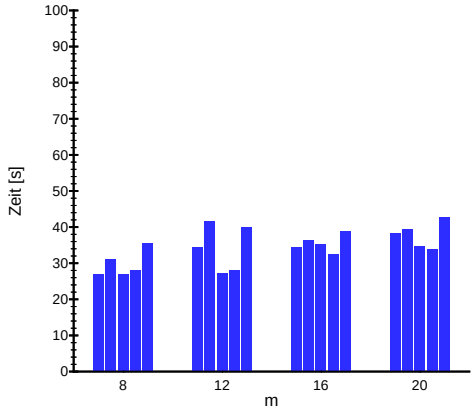
Abbildung 5.4: Rotationsströmung, $\varepsilon = 10^{-6}$, $h = \frac{1}{256}$

5.5.2 Laufzeitvergleich



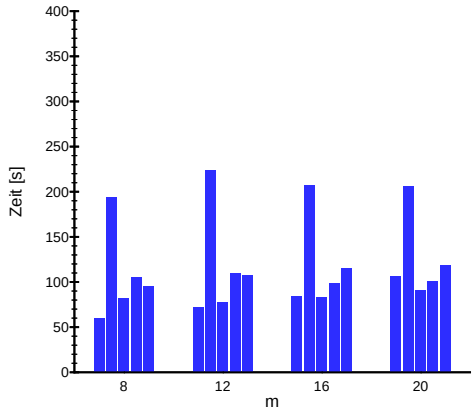
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	-	-	-	-	-
12	-	-	-	899	-
16	-	-	-	650	1022
20	-	1037	-	512	622

$n = 33 \times 33$



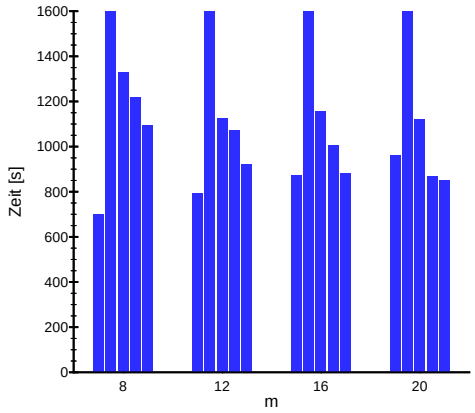
	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	1334	1386	829	637	805
12	883	902	525	402	562
16	659	669	388	295	399
20	526	514	308	233	329

$n = 65 \times 65$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	624	-	497	514	460
12	413	-	322	293	289
16	310	661	232	196	210
20	247	490	175	150	165

$n = 129 \times 129$



	Iterationen				
m	ohne	JOR	SOR	SSOR	ILU
8	1254	-	1592	1117	1057
12	824	-	867	643	562
16	617	-	596	436	360
20	492	1693	428	259	257

$n = 257 \times 257$

Kapitel 6

Zusammenfassung und Ausblick

In diesem Kapitel soll ein Fazit gezogen werden; es erfolgt eine Einschätzung der Ergebnisse und eine Betrachtung der weiteren Ansätze zur Beschleunigung der Lösung von linearen Gleichungssystemen.

Um lineare Gleichungssysteme schnell lösen zu können, benötigt man zum einen numerische Verfahren mit guten Konvergenzeigenschaften, von denen einige in Kapitel 2 vorgestellt wurden. Zum anderen braucht man geeignete Datenstrukturen, auf denen diese Algorithmen operieren (vgl. Kapitel 3). Des weiteren ist es erforderlich, diese Algorithmen mit den gegebenen Sprachmitteln einer höheren Programmiersprache effizient zu implementieren. Daß es hierbei unterschiedlich schnelle Implementierungen geben kann, wurde unter anderem in Kapitel 4 geschildert. Die dort vorgestellte Klassenbibliothek wurde anschließend im fünften Kapitel anhand eines Modellproblems einem Test unterzogen.

Aus Sicht der numerischen Mathematik ist interessant, daß zumindest bei den schweren Konvektions-Diffusions-Problemen, also bei kleinem ε und insbesondere bei sehr feinem Gitter, die Vorkonditionierung mit JOR, SOR, SSOR oder ILU nicht immer die erhoffte Wirkung zeigt. Erst bei größeren Restartlängen ist der vorkonditionierte GMRES schneller als der nichtvorkonditionierte. Die zwar vorhandene Reduktion der Iterationszahlen reicht in vielen Fällen nicht aus, um auch die Laufzeit deutlich zu senken. Dies liegt zu einem Teil auch daran, daß insbesondere bei der ILU-Vorkonditionierung die zusätzlichen Daten für den Vorkonditionierer nicht mehr in den Cache des Rechners passen. Nicht zuletzt dank der SUPG-Stabilisierung hat man bei diesen Problemen im Innern des Gebiets auch erheblich weniger Oszillationen als am Rand. Die dort auftretenden parabolischen Grenzschichten haben eine Breite von $\mathcal{O}(\sqrt{\varepsilon})$ (vgl. [RST96], Seite 184), was auch erklären würde, warum die Löserzeiten für $\varepsilon = 10^{-4}$ höher sind als bei $\varepsilon = 10^{-6}$. Für $h = 1/129$ und $h = 1/257$ liegen dann nämlich bereits etliche Gitterpunkte innerhalb der Grenzschicht, weshalb sie besser aufgelöst wird als im Fall von $\varepsilon = 10^{-6}$.

Weiterhin ist festzustellen, daß der SOR- ebenso wie der SSOR-Vorkonditionierer mit dem richtigen Relaxationsparameter ω eine durchaus ernstzunehmende Alternative zur ILU-Zerlegung darstellt. Wie im Anhang B im einzelnen dargestellt wird, reagieren diese Vorkonditionierer teilweise sehr empfindlich auf eine Änderung von ω. So macht beispielsweise bei $h = 1/256$, $\varepsilon = 10^{-6}$ und $m = 20$ der Unterschied zum zweitbesten ω beim SSOR eine Zeitspanne von 160 Sekunden aus, was in diesem Fall eine um 19% längere Laufzeit gegenüber der mit ω_{opt} bedeutet.

Eine weitere Erkenntnis ist hierbei einerseits, daß die JOR-Vorkonditionierung nahezu

unempfindlich gegenüber der Variation des Relaxationsparameters ist, und daß andererseits die Jacobi- bzw. JOR-Vorkonditionierung generell keine große Verminderung der Iterationszahl erbringt und deshalb meist sogar eine längere Laufzeit bewirkt.

Was die Vorkonditionierung anbetrifft, so wäre es sicher noch interessant, weitere der zahlreichen ILU-Varianten, wie beispielsweise ILU(p) (vgl. Bemerkung 2.41) zu implementieren. Weitere Krylov-Verfahren können ebenfalls eine weitere Beschleunigung erzielen, das CG- und das BiCGSTAB-Verfahren (vgl. z.B. [Mei99], Abschnitt 4.3.2.7) wurden bereits implementiert, aber in diesem Rahmen nicht anhand des Modellproblems getestet – das CG-Verfahren eignet sich ohnehin nur für symmetrische Probleme. Andere Krylov-Methoden wie QMRCGSTAB oder TFQMR werden gegebenenfalls in einer späteren Version von **lins** implementiert.

Ein weiterer, sehr erfolgreicher Weg bei der Lösung linearer Systeme sind die sogenannten *Mehrgitterverfahren*. Hierbei wird das Problem auch auf gröberen Gittern diskretisiert – idealerweise so, daß man das Problem auf dem gröbsten Gitter exakt lösen kann. Die Startnäherung wird auf das nächstgröbere Gitter *restringiert*, was rekursiv so lange geschieht, bis man auf einem so groben Gitter angelangt ist, daß man das Problem exakt oder zumindest mit iterativen Methoden sehr schnell lösen kann. Die Lösung eines gröberen Gitters wird dann geeignet auf das nächstfeinere *prolongiert* und dort mit einem einfachen Iterationsverfahren wie Jacobi oder Gauß-Seidel *geglättet*.

Ein relativ neuer und vielversprechender Ansatz sind darüber hinaus *algebraische Mehrgitterverfahren*. Hier löst man sich von der geometrischen Struktur des Problems und generiert die kleineren Hilfsprobleme aus der Matrix selbst. Dadurch sind diese Verfahren unabhängig von konkreten Diskretisierungen und können als allgemeines Verfahren zur Lösung von linearen Gleichungssystemen eingesetzt werden. In [RS87] werden hierfür auch Konvergenzaussagen im Falle symmetrischer M-Matrizen bewiesen.

Was den Standpunkt der Informatik anbetrifft, so zeigt sich, daß es auch mit den objektorientierten Mitteln von C++ möglich ist, eine effiziente Implementierung numerischer Algorithmen zu leisten, was auch ein wesentliches Ziel dieser Arbeit war. Die Schwierigkeit besteht darin, auf der einen Seite eine High-Level Struktur zu implementieren (übersichtlicher Code, Wahrung von objektorientierten Paradigmen wie Kapselung) und auf der anderen Seite dem Compiler genug Informationen zu überlassen, anhand derer er den Code optimieren kann. Der Zugriff über Iteratoren auf die Matrix-Container und die vorsichtige Verwendung von virtuellen Funktionen sorgen bei **lins** dafür, daß der Compiler dies an vielen Stellen auch wirklich tut. Jedoch stehen sich diese beiden Forderungen (die nach „Schönheit“ und nach Effizienz) prinzipiell diametral gegenüber.

Auch diese Klassenbibliothek ist an vielen Stellen noch nicht optimal. Beispielsweise erweist es sich als sinnvoll, möglichst viele Eigenschaften einer Klasse als Template-Parameter festzulegen. Da der Wert bzw. Typ eines Template-Parameters zur Übersetzzeit bekannt sein muß, kann der Compiler dann auch entsprechende Optimierungen vornehmen.

Ein weiterer Ansatzpunkt ist die Optimierung von Vektor-Ausdrücken mit Hilfe von *Expression Templates* (siehe z.B. [Vel95]). Um beispielsweise einen Ausdruck wie

$$y = v - w + \alpha x, \quad v, w, x, y \in \mathbf{R}^n, \alpha \in \mathbf{R} \tag{6.1}$$

effizient abzuarbeiten, müssen die Komponenten von y in *einer* Schleife berechnet werden. Es reicht nicht, nur die einfachen Grundoperationen wie $+$, $-$ und die Skalarmultiplikation

zu implementieren, da dann erstens temporäre Vektoren benötigt und zweitens mindestens drei Schleifen vom Compiler generiert würden. Im bisherigen Fortran- und C-Stil wurde deshalb für jede Art von Vektorausdruck eine eigene Funktion geschrieben. Dies ist in blanc und auch noch in dieser Bibliothek der Fall (siehe Tabelle 4.6). Expression Templates machen dies nun überflüssig und ermöglichen es dem Compiler, nun *jede* beliebige Art von Vektorausdrücken so zu optimieren, daß keine temporären Objekte benötigt werden und das Ergebnis trotzdem in einem Durchgang berechnet wird, obwohl hierfür nur die grundlegenden Operatoren wie `+`, `-`, `*` implementiert werden. Hinzu kommt, daß man diese Vektorausdrücke gewissermaßen in „Klartextschreibweise“ im Programmtext verwenden kann. Den Ausdruck (6.1) etwa würde man dann als

```
DenseVector<double> v(n), w(n), x(n), y(n);
double alpha;
...
y = v - w + (alpha * x);
```

schreiben, was die Lesbarkeit und damit die Wartbarkeit des Codes ungemein erhöht.

Anhang A

Klassendiagramme

A.1 Matrixklassen

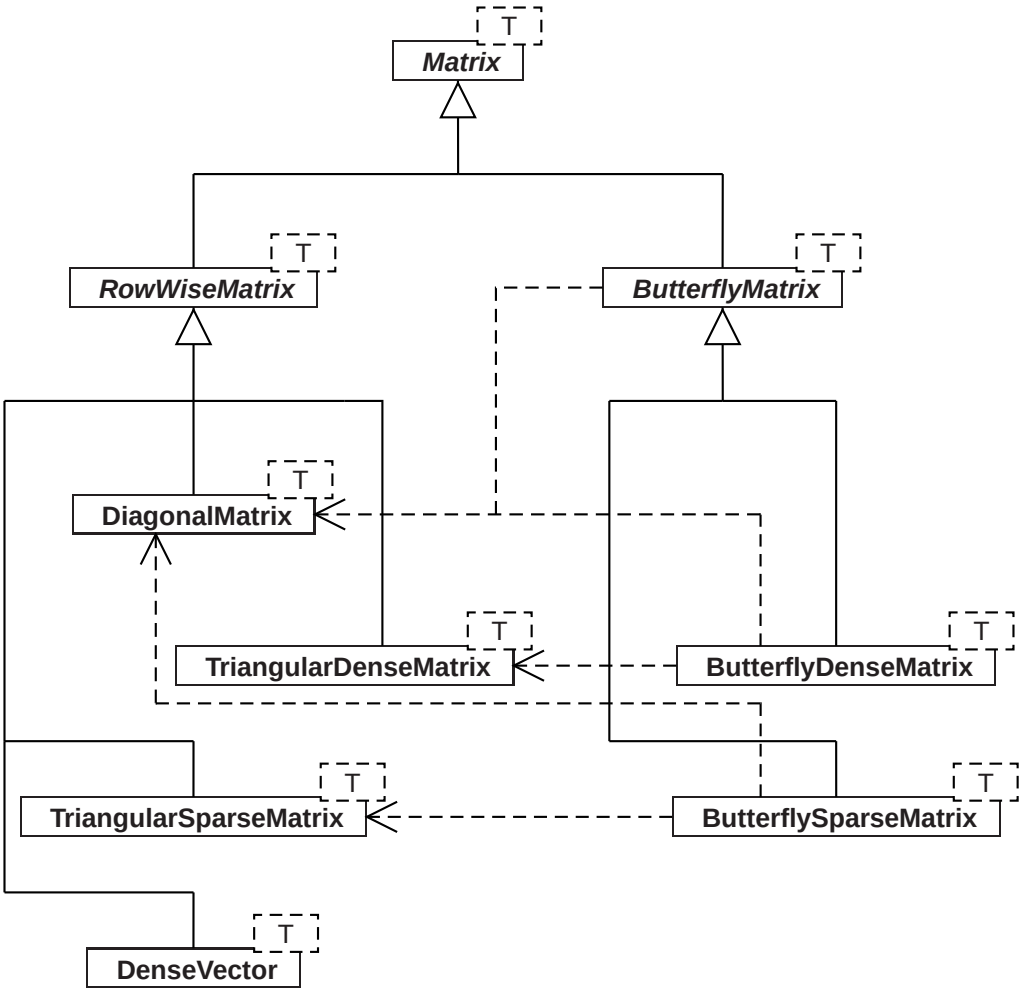


Abbildung A.1: Klassendiagramm aller Matrix- und Vektorklassen

A.2 Krylov-Unterraum-Verfahren

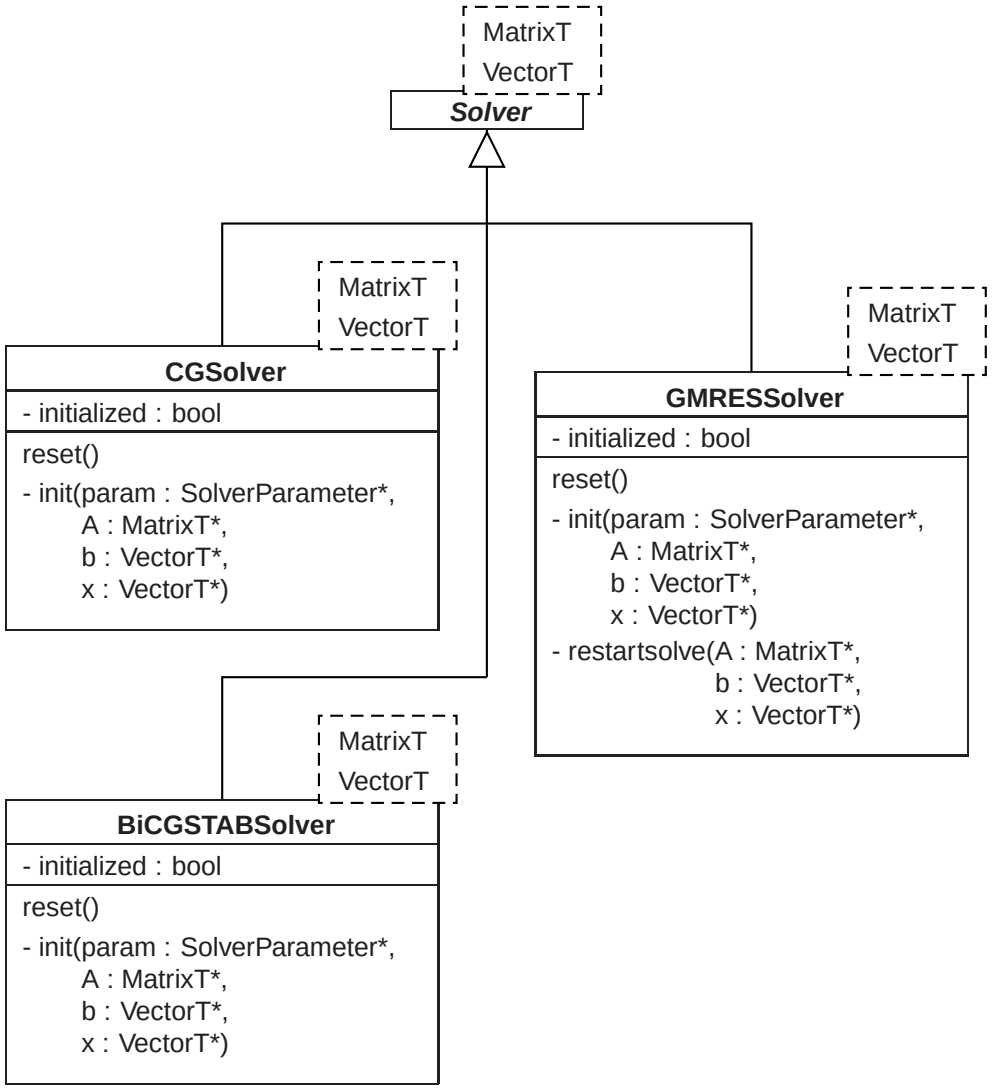


Abbildung A.2: Klassendiagramm der Krylovverfahren

A.3 Splitting-Methoden

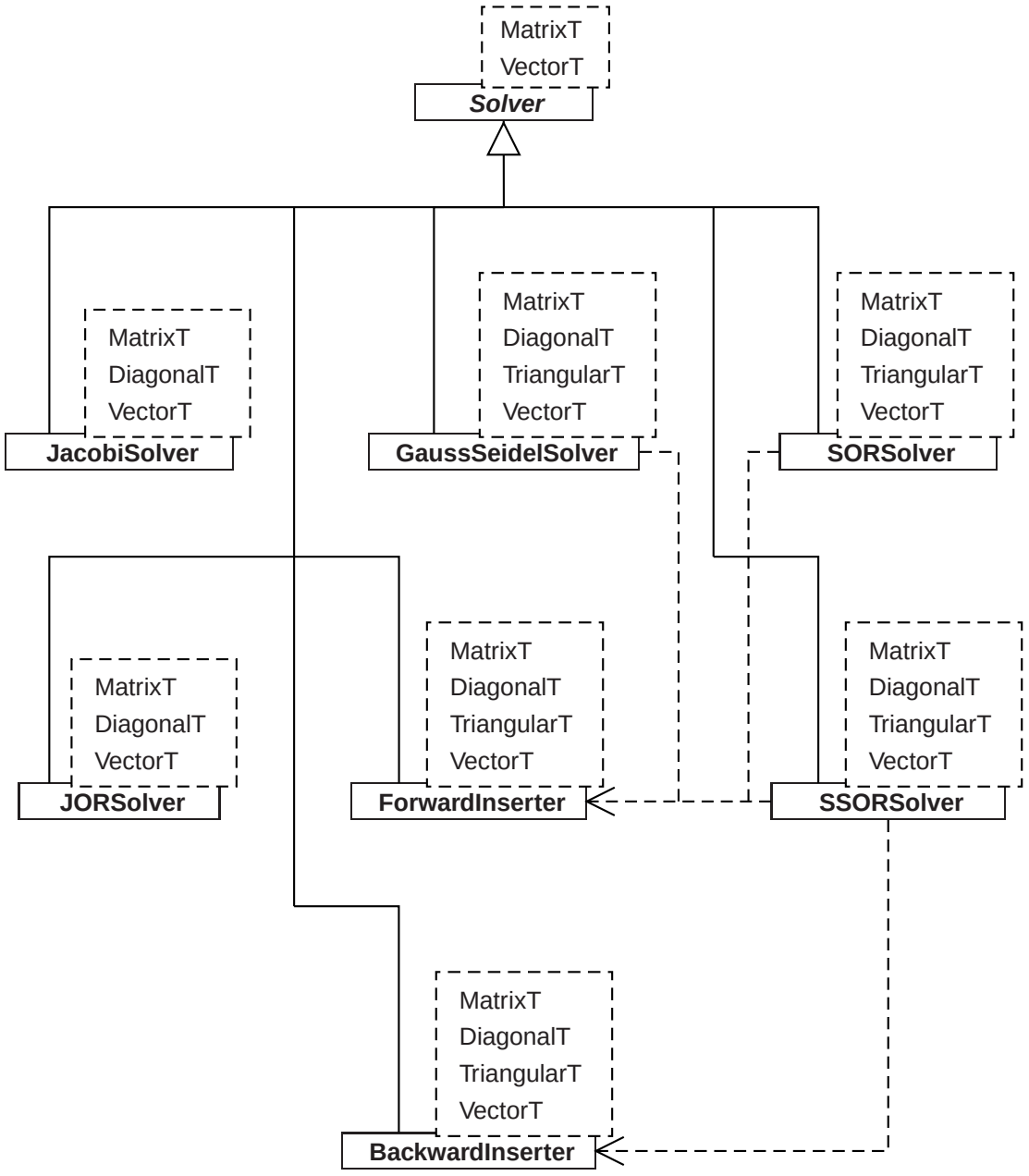


Abbildung A.3: Klassendiagramm der Splittingverfahren

A.4 Splitting-Vorkonditionierer

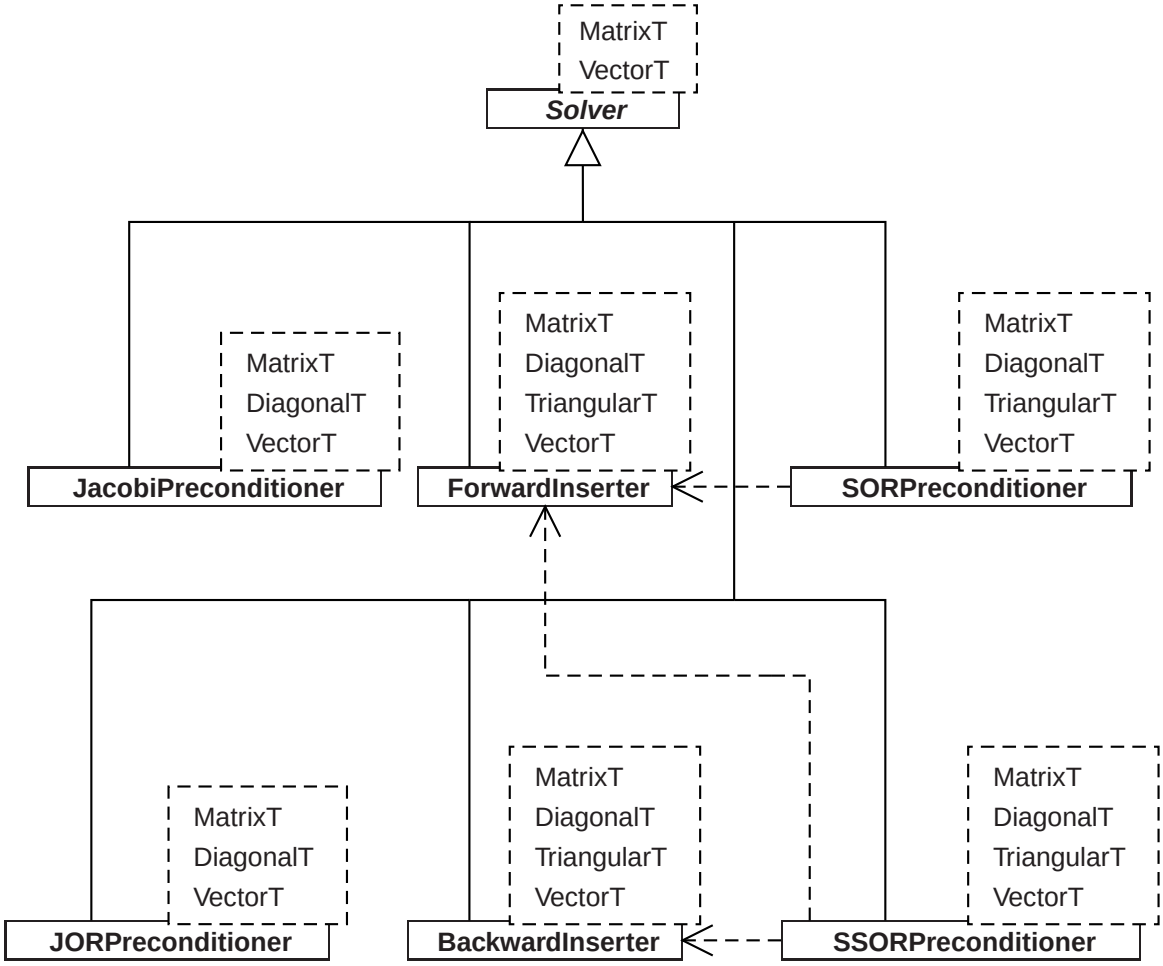


Abbildung A.4: Klassendiagramm der Splitting-Vorkonditionierer

A.5 Sonstige Vorkonditionierer

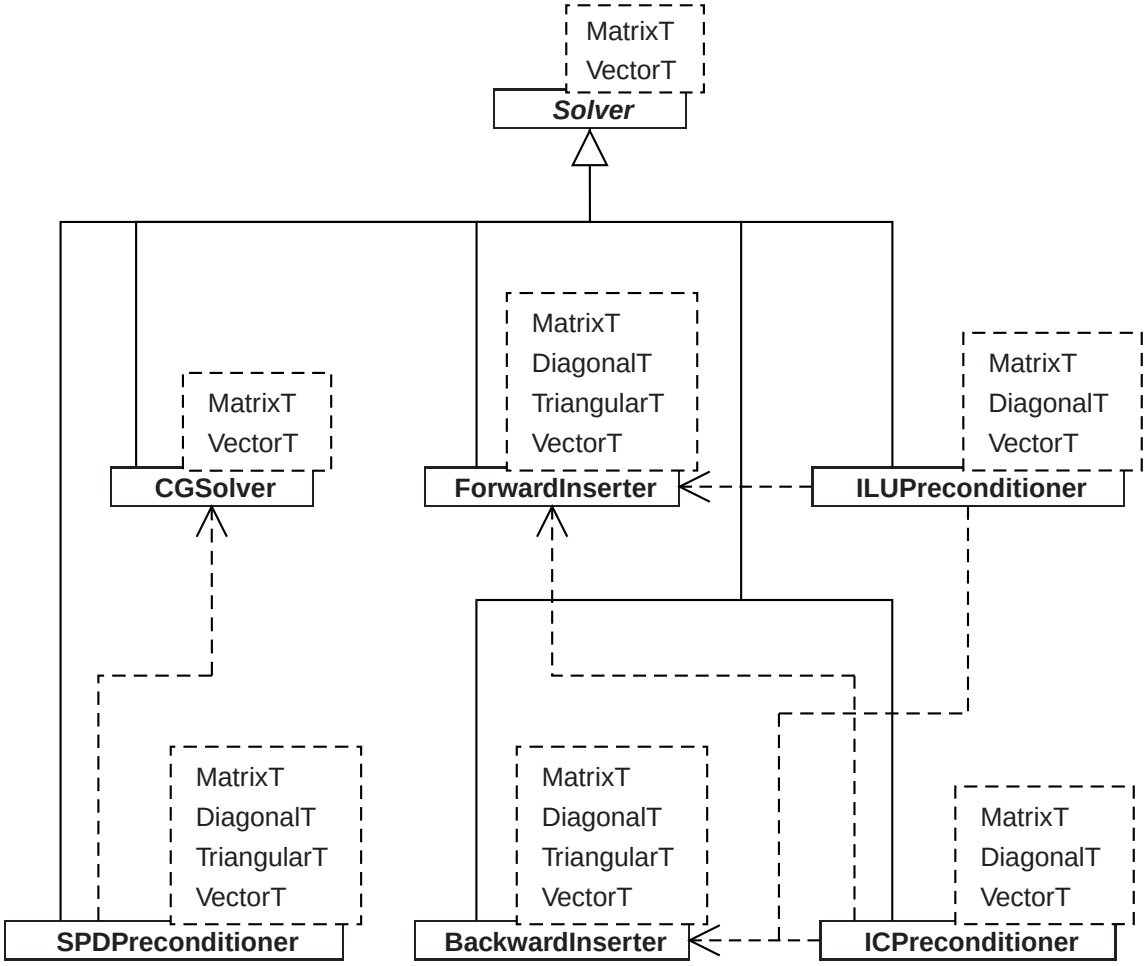


Abbildung A.5: Klassendiagramm der sonstigen Vorkonditionierer

Anhang B

Empirische Bestimmung von ω_{opt}

Wie bereits in [Pri96] festgestellt wurde, hängt der Wirkungsgrad eines relaxierten Splitting-Vorkonditionierers oft entscheidend von der Wahl des Relaxationsparameters ω ab.

Wie aus den Sätzen 2.10 und 2.14 aus Kapitel 2 hervorgeht, läßt sich unter gewissen Bedingungen, beispielsweise, wenn A symmetrisch positiv definit oder konsistent geordnet ist, ein optimaler Relaxationsparameter für diverse Splitting-Verfahren angeben. Bei vielen Problemen, wie sie zum Beispiel bei der Diskretisierung von singulär gestörten Konvektions-Diffusionsgleichungen auftauchen, sind diese Bedingungen jedoch meist nicht erfüllt, da hier der konvektive, also nicht-symmetrische Anteil dominiert. Da hierzu noch zu wenige theoretische Resultate vorliegen (in [BR99] werden zumindest Konvergenzaussagen für das JOR-Verfahren bewiesen), ist man weitestgehend auf experimentelles Vorgehen angewiesen.

In [Pri96], Abschnitt 4.4.1, wurden für das SOR- und SSOR-*Verfahren* anhand verschiedener Probleme und Gittergrößen die jeweiligen optimalen Relaxationsparameter experimentell bestimmt. Jedoch gilt es nicht als gesichert, daß diese auch die optimalen Werte für die jeweiligen *Vorkonditionierer* sind. Deshalb wird an dieser Stelle das GMRES(m)-Verfahren mit den Vorkonditionierern JOR, SOR und SSOR für das in (5.1) beschriebene Modellproblem getestet. Im folgenden wurde dieses Problem für $\varepsilon = 10^{-k}, k = 0, 2, 4, 6$ untersucht, wobei sich die Betrachtungen auf den numerisch interessantesten, weil schwierigsten Fall $c \equiv 0$ beschränken.

Es wurde jeweils mit einem strukturierten Gitter auf dem Einheitsquadrat, für $h \in \{\frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}\}$ gerechnet. Die hieraus resultierenden Matrixdimensionen $n = (\frac{1}{h} + 1)^2$ sind dabei von links nach rechts in aufsteigender Reihenfolge angegeben. Das Konvergenzkriterium bestand in der Reduktion des (vorkonditionierten) Residuums um 10^{-6} in der euklidischen Norm. In den Diagrammen ist jeweils nach oben die Anzahl der Restarts abgetragen, die das GMRES(m)-Verfahren bis zur Erfüllung dieses Kriteriums benötigte. Es sind jeweils 4 Restartlängen ($m = 8, 12, 16, 20$) eingezeichnet :

- $m = 8$: gepunktete Linie,
- $m = 12$: Strich-Punkt-Linie,
- $m = 16$: gestrichelte Linie,
- $m = 20$: durchgezogene Linie.

Das jeweils oberste Diagramm zeigt die Ergebnisse der JOR-Vorkonditionierung, das mittlere die SOR- und das unterste die SSOR-Vorkonditionierung. Auf der x-Achse wurde der Relaxationsparameter ω im Intervall $[0.05, 1.95] \subset (0, 2)$ in 39 äquidistanten Schritten variiert. Es ist also

$$\omega \in \{x \in \mathbf{R} \mid x = i \cdot 0.05, i = 1, \ldots, 39\}.$$

In den Tabellen wurden zu jedem Problem noch einmal die optimalen Relaxationsparameter und die zugehörigen Restarts explizit aufgeführt. Manche der Rechnungen führten innerhalb der vorgegebenen Zahl der Restarts nicht zur Konvergenz, was in den Tabellen durch einen Strich gekennzeichnet wurde.

Es zeigt sich, daß bei großen Werten für ε eine Überrelaxation (also $\omega > 1$) des SOR und SSOR zum Erfolg führt, während bei kleiner werdendem ε eher eine Unterrelaxation ($\omega < 1$) zu empfehlen ist. Beide Vorkonditionierer sind teilweise sehr sensibel gegenüber einer Änderung von ω, insbesondere bei kleinem ε und zu groß gewähltem ω führte dies häufig zur Divergenz des Verfahrens.

Die Relaxierung des Jacobi-Vorkonditionierers erwies sich jedoch in der Praxis als relativ unbrauchbar. Verbesserungen der benötigten Restarts zur Konvergenz sind bei verändertem ω allenfalls marginal. Etwas anderes war sicherlich auch nicht zu erwarten, da ja der JOR- gegenüber dem Jacobi-Vorkonditionierer lediglich mit einer reellen Zahl skaliert ist. Die Ergebnisse wurden jedoch der Vollständigkeit halber mit aufgenommen.

B.1 Rotationsströmung, $\varepsilon = 1$

GMRES(m) mit JOR-Vorkonditionierung

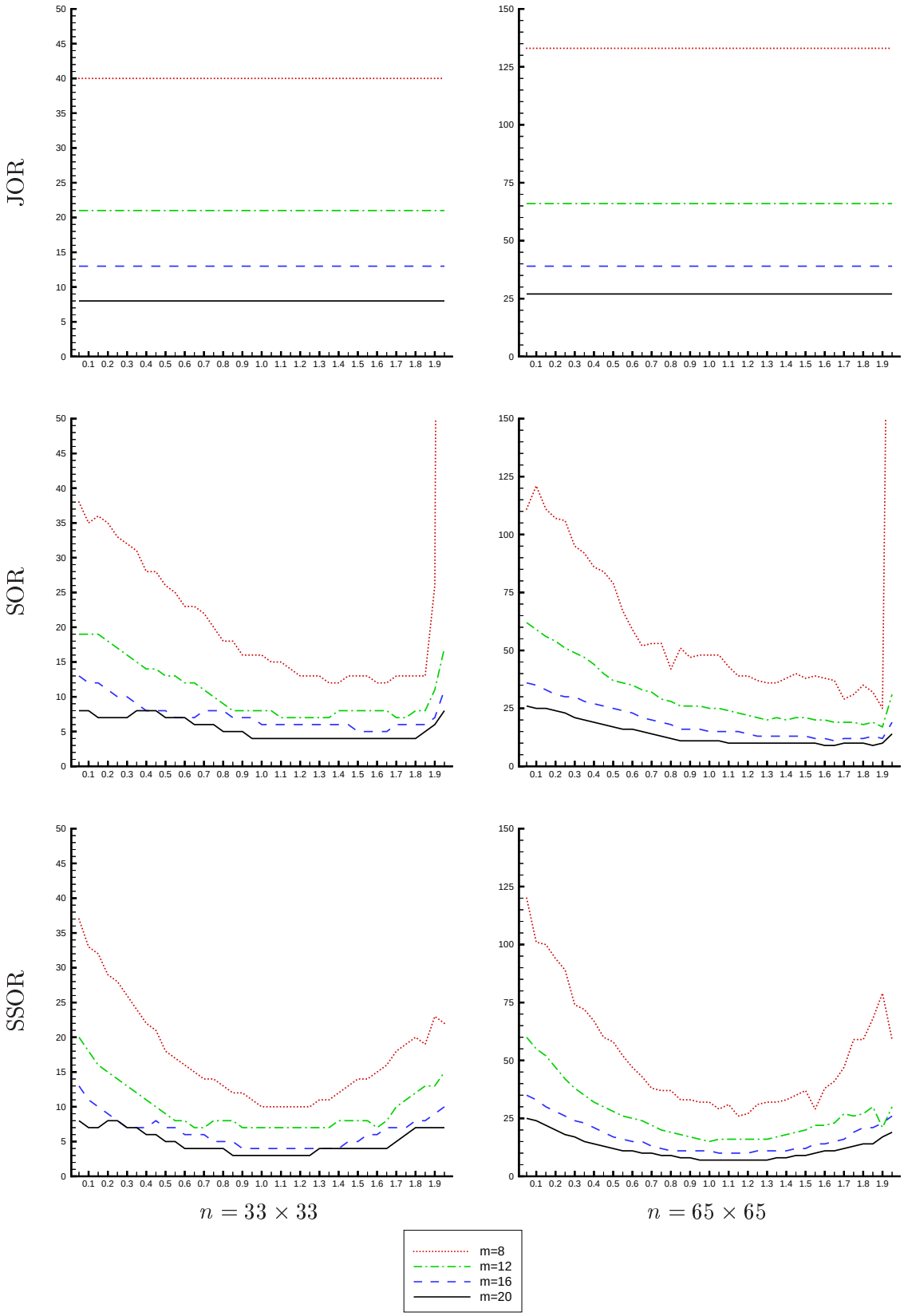
	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.00	40	1.00	133	1.00	450	1.00	1392
12	1.00	21	1.00	66	1.00	220	1.00	756
16	1.00	13	1.00	39	1.00	129	1.00	446
20	1.00	8	1.00	27	1.00	86	1.00	289

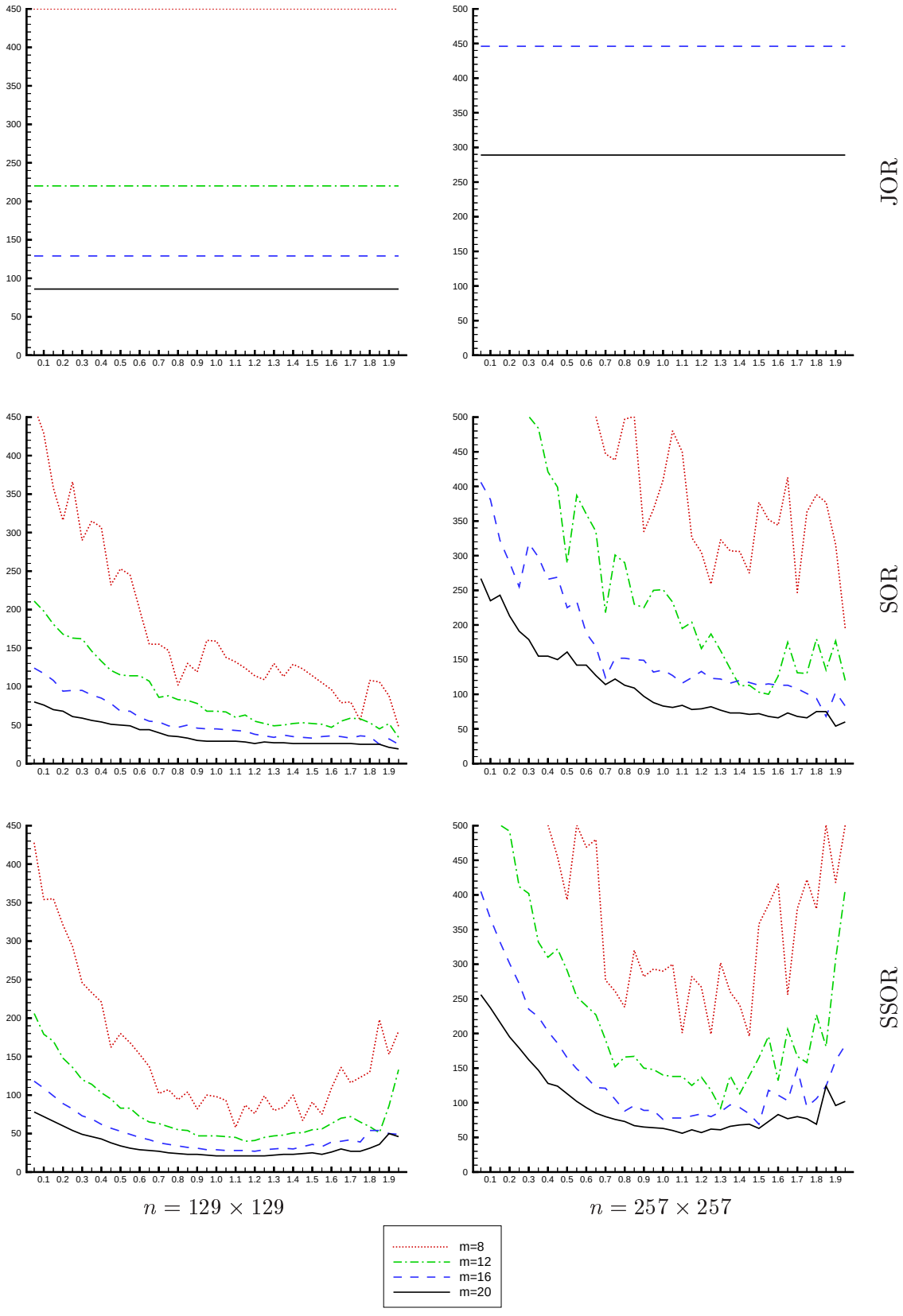
GMRES(m) mit SOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.35	12	1.90	25	1.95	48	1.95	195
12	1.10	7	1.90	17	1.95	34	1.55	100
16	1.50	5	1.65	11	1.85	25	1.85	68
20	1.00	4	1.60	9	1.95	19	1.90	54

GMRES(m) mit SSOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.00	10	1.15	26	1.10	58	1.45	196
12	1.00	7	1.00	15	1.15	40	1.30	92
16	1.00	4	1.05	10	1.20	27	1.50	69
20	1.00	3	1.00	7	1.00	21	1.10	56


JOR
SOR
SSOR
$n = 33 \times 33$
$n = 65 \times 65$
m=8
m=12
m=16
m=20


JOR
SOR
SSOR
$n = 129 \times 129$
$n = 257 \times 257$
m=8
m=12
m=16
m=20

B.2 Rotationsströmung, $\varepsilon = 10^{-2}$

GMRES(m) mit JOR-Vorkonditionierung

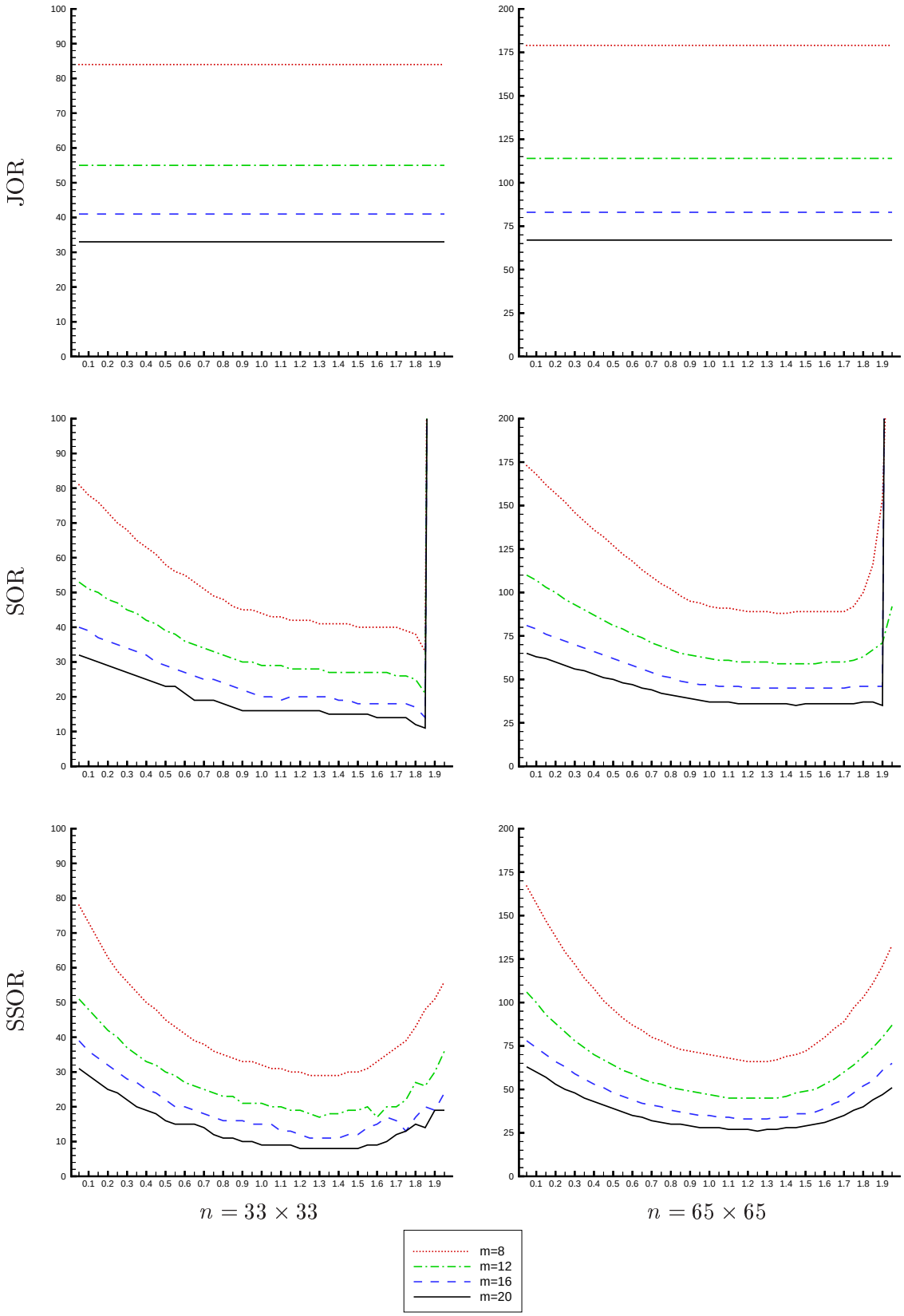
	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.00	84	1.00	179	1.00	379	1.00	940
12	1.00	55	1.00	114	1.00	225	1.00	482
16	1.00	41	1.00	83	1.00	163	1.00	321
20	1.00	33	1.00	67	1.00	128	1.00	242

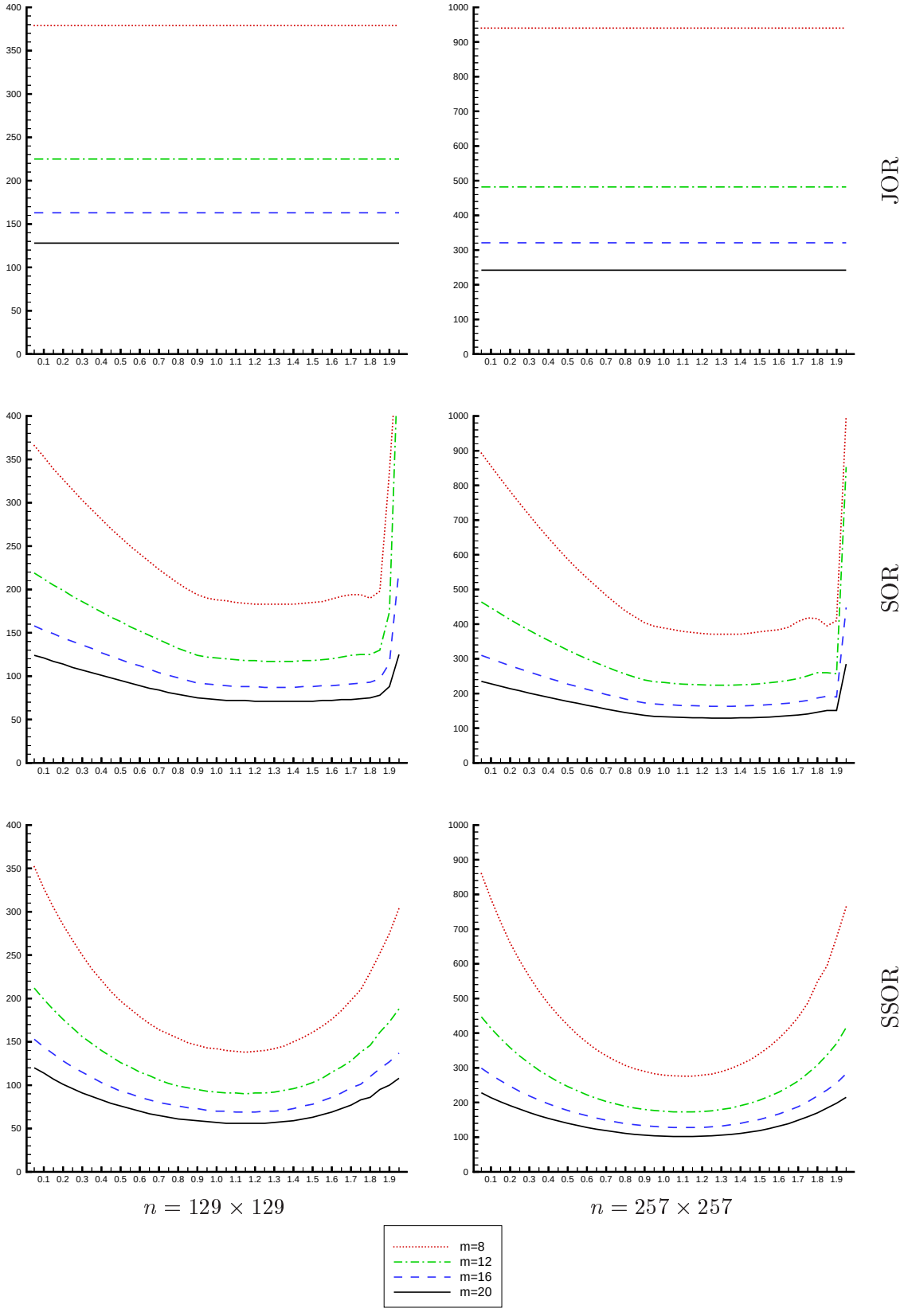
GMRES(m) mit SOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.85	33	1.35	88	1.20	183	1.25	371
12	1.85	21	1.35	59	1.25	117	1.25	224
16	1.85	14	1.20	45	1.25	87	1.25	163
20	1.85	11	1.45	35	1.20	71	1.25	129

GMRES(m) mit SSOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	1.25	29	1.20	66	1.15	138	1.10	276
12	1.30	17	1.10	45	1.15	90	1.05	173
16	1.25	11	1.15	33	1.10	69	1.05	128
20	1.20	8	1.25	26	1.05	56	1.05	102


JOR
SOR
SSOR
$n = 33 \times 33$
$n = 65 \times 65$
m=8
m=12
m=16
m=20


JOR
SOR
SSOR
$n = 129 \times 129$
$n = 257 \times 257$
m=8
m=12
m=16
m=20

B.3 Rotationsströmung, $\varepsilon = 10^{-4}$

GMRES(m) mit JOR-Vorkonditionierung

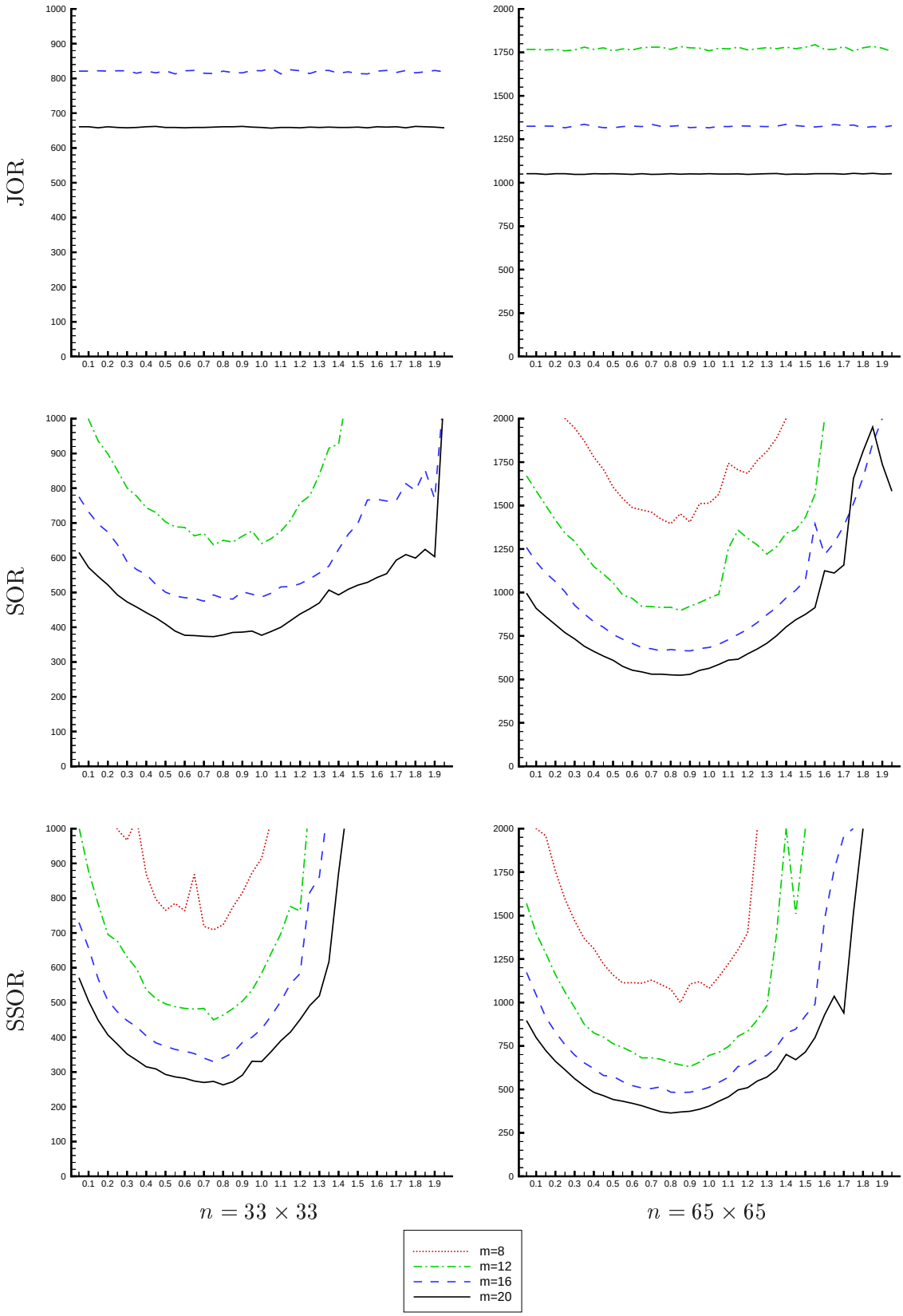
	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	-	-	-	-	-	-	-	-
12	-	-	1.95	1755	1.00	1794	1.00	1282
16	0.55	813	1.00	1316	0.95	1361	1.00	944
20	1.05	657	0.70	1048	1.45	1083	1.00	750

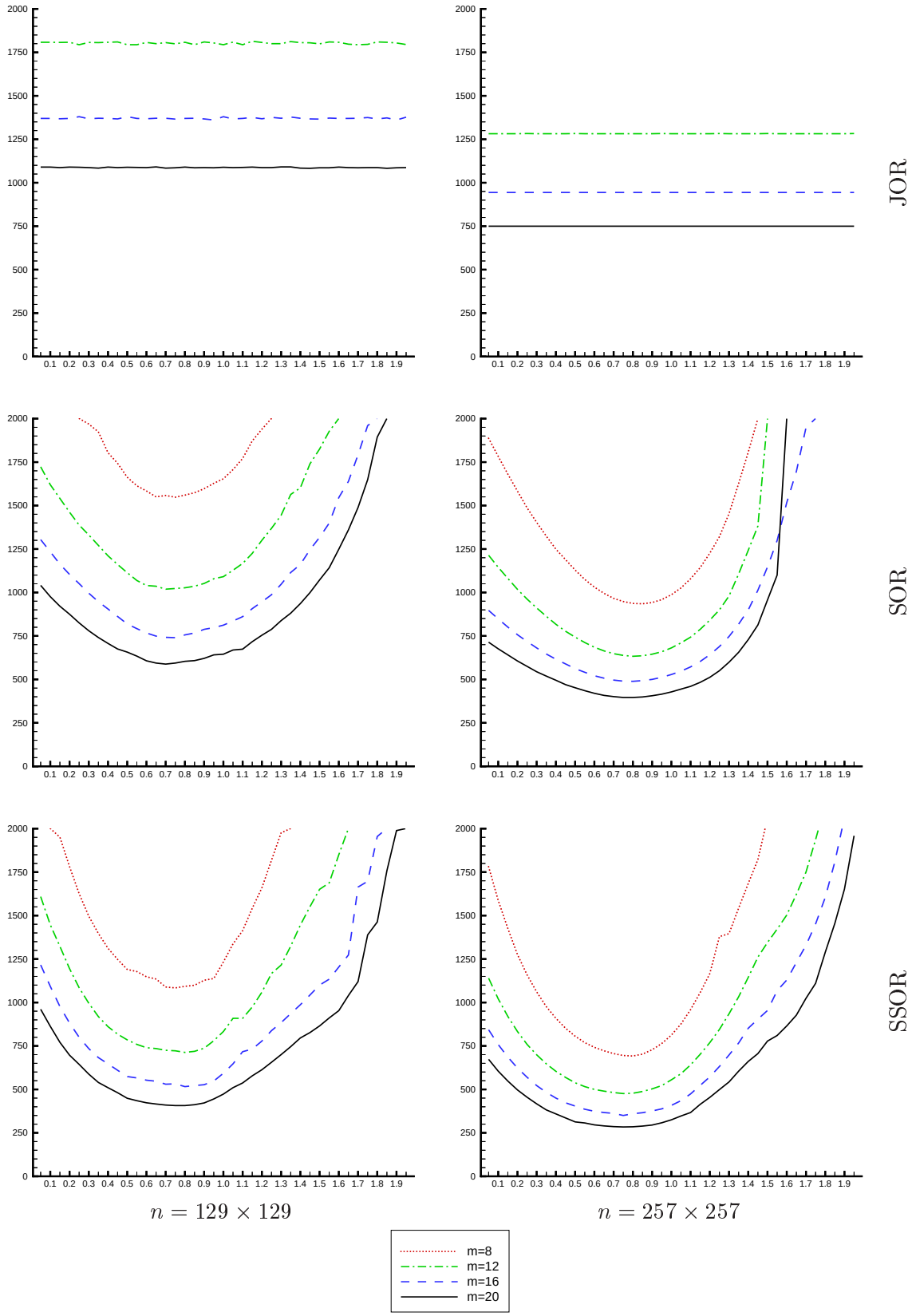
GMRES(m) mit SOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	0.75	1047	0.80	1396	0.75	1548	0.85	935
12	0.75	637	0.85	896	0.70	1019	0.80	633
16	0.70	475	0.90	664	0.75	740	0.80	489
20	0.75	373	0.85	524	0.70	588	0.80	396

GMRES(m) mit SSOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	0.75	709	0.85	999	0.75	1084	0.80	692
12	0.75	450	0.90	631	0.80	713	0.75	476
16	0.75	330	0.85	482	0.80	516	0.75	350
20	0.80	263	0.80	364	0.80	407	0.75	284


JOR
SOR
SSOR
$n = 33 \times 33$
$n = 65 \times 65$
m=8
m=12
m=16
m=20


JOR
SOR
SSOR
$n = 129 \times 129$
$n = 257 \times 257$
m=8
m=12
m=16
m=20

B.4 Rotationsströmung, $\varepsilon = 10^{-6}$

GMRES(m) mit JOR-Vorkonditionierung

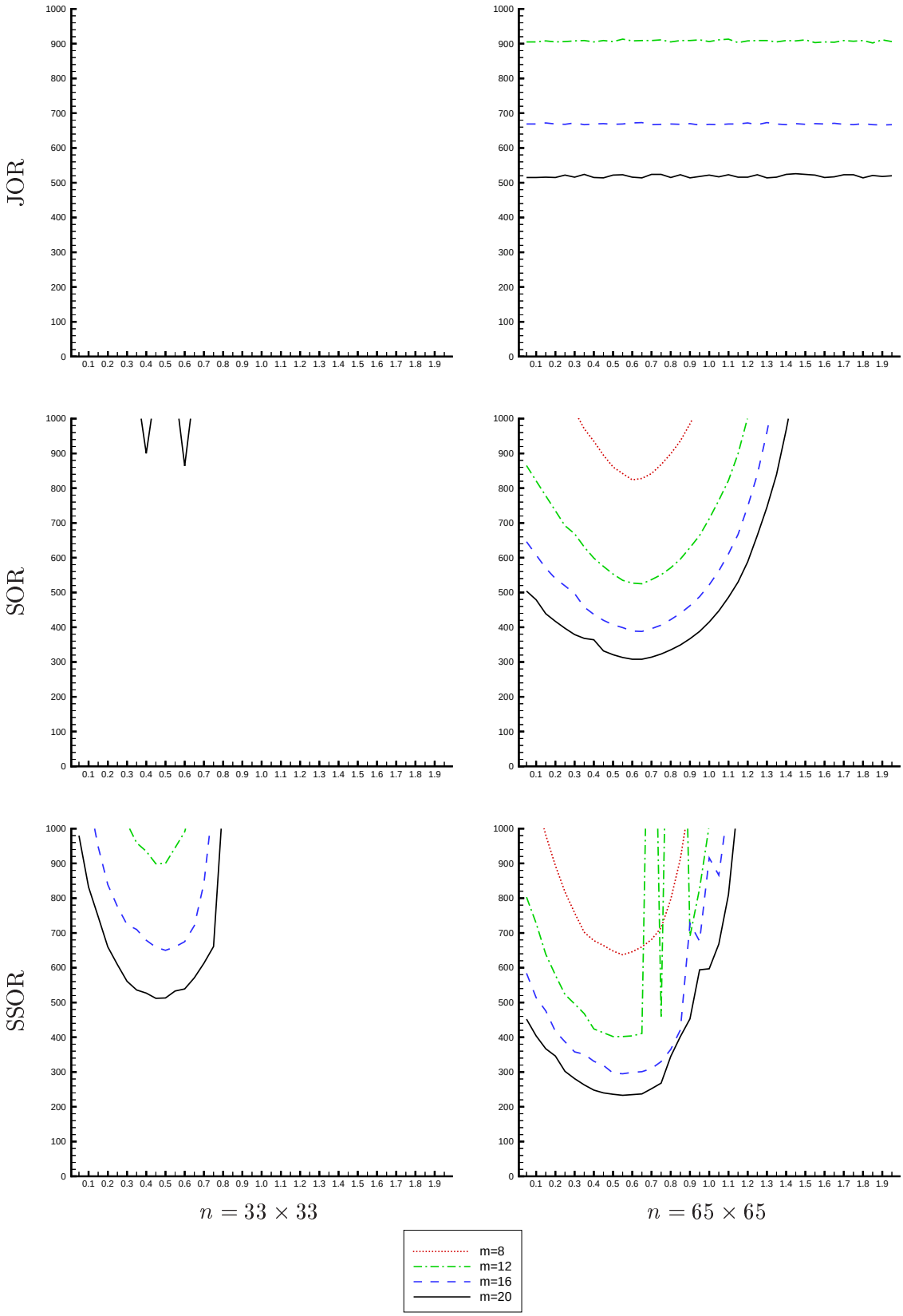
	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	-	-	1.75	1386	-	-	-	-
12	-	-	1.85	902	-	-	-	-
16	-	-	0.95	666	0.55	661	-	-
20	0.55	1037	0.90	514	0.75	490	1.95	1596

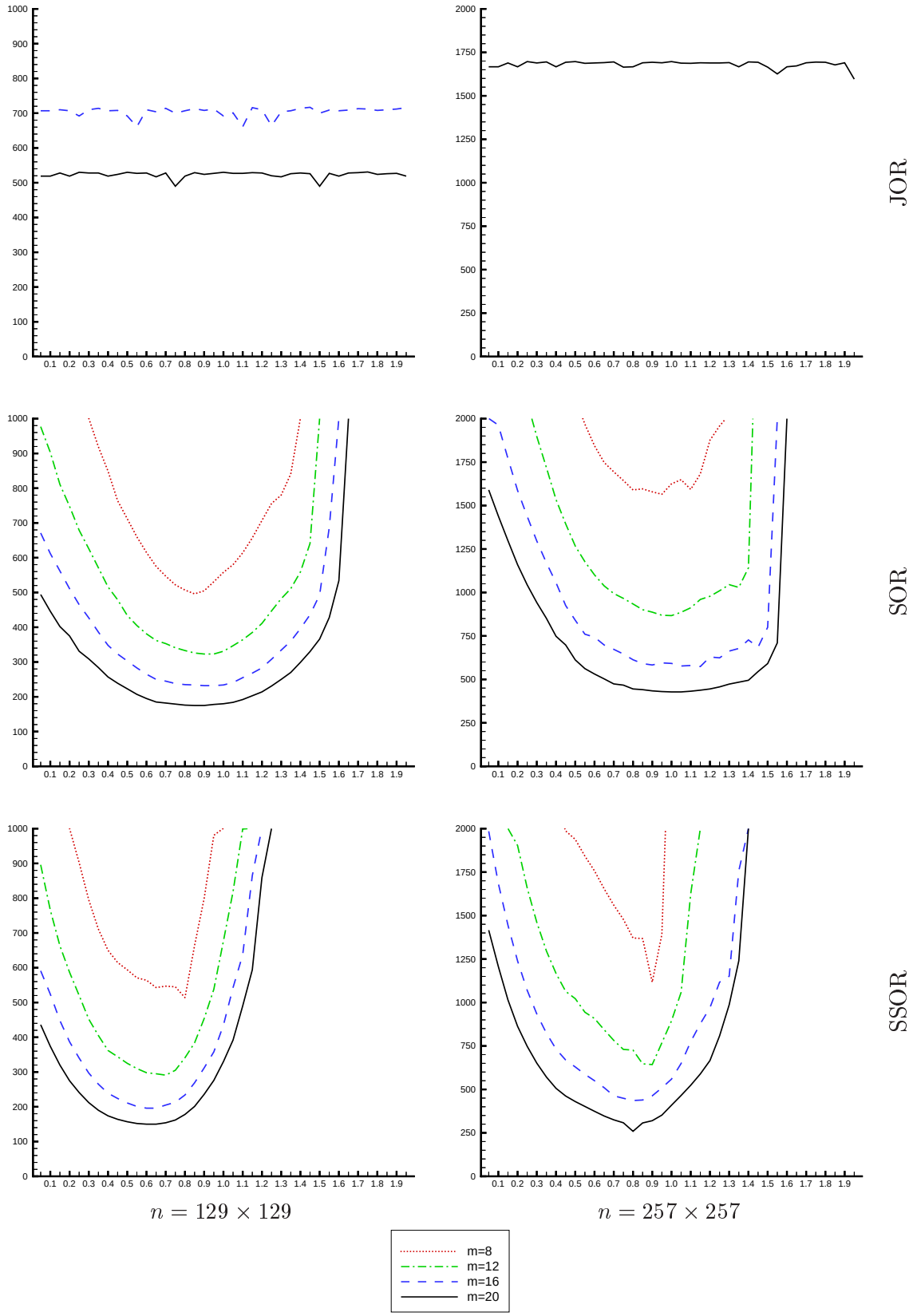
GMRES(m) mit SOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	-	-	0.60	824	0.85	496	0.95	1565
12	-	-	0.65	525	0.95	323	1.00	867
16	-	-	0.65	388	0.95	232	1.15	574
20	0.60	864	0.65	308	0.90	175	1.00	428

GMRES(m) mit SSOR-Vorkonditionierung

	$n = 33 \times 33$		$n = 65 \times 65$		$n = 129 \times 129$		$n = 257 \times 257$	
m	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen	ω_{opt}	Iterationen
8	-	-	0.55	637	0.80	514	0.90	1117
12	0.45	899	0.55	402	0.70	291	0.90	643
16	0.50	650	0.55	295	0.65	196	0.80	436
20	0.45	512	0.55	233	0.65	150	0.80	259


JOR
SOR
SSOR
$n = 33 \times 33$
$n = 65 \times 65$
m=8
m=12
m=16
m=20


JOR
SOR
SSOR
$n = 129 \times 129$
$n = 257 \times 257$
m=8
m=12
m=16
m=20

Anhang C

Testplattformen

C.1 Compaq AlphaStation XP1000

Prozessor	Alpha 21264A (EV67)
Taktfrequenz	667 MHz
L1-Cache	64 KB + 64 KB
L2-Cache	4 MB
RAM	1280 MB
RAM-Typ	100 Mhz ECC SDRAM
Betriebssystem	Compaq Tru64 UNIX
Compiler	g++ v2.95.2

Tabelle C.1: Alpha AXP Konfiguration

C.2 Intel PIII

Prozessor	Intel Pentium III
Taktfrequenz	450 MHz
L1-Cache	16 KB + 16 KB
L2-Cache	512 KB
RAM	192 MB
RAM-Typ	100 MHz SDRAM
Betriebssystem	SuSE Linux 6.4
Compiler	g++ v2.95.2

Tabelle C.2: Intel Konfiguration

Verzeichnis der Algorithmen

Literaturverzeichnis

[Alt92] Alt, Hans Wilhelm: *Lineare Funktionalanalysis.* Springer, Berlin - Heidelberg - New York - Tokyo, 1992.

[AO+99] Auge, Andreas, Frank-Christian Otto et al.: *ParallelNS User's Guide*, 1999.

[BB85] Bunse, Wolfgang und Bunse-Gerstner, Angelika: *Numerische lineare Algebra.* B. G. Teubner, Berlin - Heidelberg - New York - Tokyo, 1985.

[BBC+94] Barrett, R., M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine und H. Van der Vorst: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, 1994.
`ftp://ftp.netlib.org/templates/templates.ps`.

[BR99] Bey, Jürgen und Arnold Reusken: *On the Convergence of Basic Iterative Methods for Convection-Diffusion Equations.* Numerical Linear Algebra With Applications, 6:329–352, 1999.

[BRJ99] Booch, Grady, Rumbaugh, Jim und Jacobson, Ivar: *Das UML-Benutzerhandbuch.* Addison-Wesley-Longman, Bonn, 1999.

[Dem97] Demmel, James W.: *Applied Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[Die96] Diener, Immo: *Numerische Mathematik I.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Vorlesungsskript, Wintersemester 1995/1996.

[DLH00] Dutto, Laura C., Claude Y. Lepage und Wagdi G. Habashi: *Effect of the storage format of sparse linear systems on parallel CFD computations.* Comput. Methods Appl. Mech. Engrg., 188:441–453, 2000.

[GR94] Grossmann, Christian und Roos, Hans-Görg.: *Numerik partieller Differentialgleichungen.* B. G. Teubner, Stuttgart, 1994.

[Gre97] Greenbaum, Anne: *Iterative Methods for Solving Linear Systems.* SIAM, Philadelphia, 1997.

[Hac93] Hackbusch, Wolfgang: *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* B. G. Teubner, Stuttgart, 1993.

[HY81] Hageman, Louis A. und Young, David M.: *Applied Iterative Methods.* Academic Press, New York - London, 1981.

[Kno99] Knopp, Tobias: *Eine stabilisierte Finite-Elemente-Methode für das k/ε-Turbulenzmodell der inkompressiblen und nichtisothermen Navier-Stokes-Gleichungen.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Diplomarbeit, 1999.

[Koh00] Kohlhammer, Joachim: *Erzeugung und Verfeinerung von Finite-Elemente-Triangulierungen.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Diplomarbeit, 2000.

[Lub98a] Lube, Gert: *Diskretisierungsverfahren für partielle Differentialgleichungen I.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Vorlesungsskript, 1998.

[Lub98b] Lube, Gert: *Lineare Funktionalanalysis.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Vorlesungsskript, 1998.

[Mei99] Meister, Andreas: *Numerik linearer Systeme.* Friedr. Vieweg & Sohn, Braunschweig - Wiesbaden, 1999.

[ML99] Müller, Susann und Lube, Gert: *On SPD-preconditioned iterative methods applied to a stabilized Galerkin method.* In: *Proceedings of the Third European Conference on Numerical Mathematics and Advanced Applications (ENUMATH 99).* World Scientific, 1999.

[Mv77] Meijerink, J. A. und van der Vorst, H. A.: *An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix.* Mathematics of Computation, 31(137):148–162, 1977.

[Pri96] Priesnitz, Andreas P.: *Untersuchung iterativer Lösungsverfahren am Beispiel diskretisierter Konvektions-Diffusions-Reaktions-Gleichungen.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Diplomarbeit, 1996.

[RS87] Ruge, J. W. und K. Stüben: *Algebraic Multigrid.* In: McCormick, Stephen F. (Herausgeber): *Multigrid Methods*, Frontiers in Applied Mathematics. SIAM, Philadelphia, 1987.

[RST96] Roos, Hans-Görg, Stynes, Martin und Tobiska, Lutz: *Numerical Methods for Singularly Pertubated Differential Equations.* Springer, Berlin - Heidelberg - New York - Tokyo, 1996.

[Saa94] Saad, Youcef: *SPARSKIT : a basic tool kit for sparse matrix computations.* Technischer Bericht, Department of Computer Science and Engineering, University of Minnesota, 1994.
`http://www.cs.umn.edu/Research/arpa/SPARSKIT/paper.ps`.

[Saa96] SAAD, YOUCEF: *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[Sch00] SCHILLING, HEIKO: *Flußorientierte Numerierungsstrategien zur Vorkonditionierung konvektionsdominanter Probleme*. Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Diplomarbeit, 2000.

[Sie99] SIEK, JEREMY G.: *A Modern Framework for Portable High Performance Numerical Linear Algebra*. Masters Thesis, University of Notre Dame, Indiana, Department of Computer Science and Engineering, Notre Dame, Indiana, 1999. `http://www.lsc.nd.edu/downloads/research/mtl/papers/thesis.pdf`.

[SS86] SAAD, YOUCEF und M. H. SCHULTZ: *GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems*. SIAM J. Sci. Statist. Comput., 7:856–869, 1986.

[Sta97] STARKE, GERHARD: *Field-of-values analysis of preconditioned iterative methods for nonsymmetric elliptic problems*. Numerische Mathematik, 78:103–117, 1997.

[Str98] STROUSTRUP, BJARNE: *Die C++ Programmiersprache*. Addison-Wesley-Longman, Bonn, 1998.

[SW93a] SAAD, YOUCEF und KESHENG WU: *DQGMRES : a Quasi - minimal residual algorithm based on incomplete orthogonalization*. Technischer Bericht, Computer Science Department, University of Minnesota, 1993. `ftp://ftp.cs.umn.edu/dept/users/saad/reports/FILES/umsi-93-131.ps.gz`.

[SW93b] SCHABACK, ROBERT und WERNER, HELMUT: *Numerische Mathematik*. Springer, Berlin - Heidelberg - New York - Tokyo, 1993.

[Swi99] SWITZER, ROBERT: *Softwaretechnologie*. Georg-August-Universität Göttingen, Mathematisches Institut, Vorlesungsskript, Wintersemester 1998/1999.

[Vel95] VELDHUIZEN, TODD L.: *Expression templates*. C++ Report, 7(5):26–31, Juni 1995. Reprinted in C++ Gems, ed. Stanley Lippman `http://extreme.indiana.edu/~tveldhui/papers/cppworld.ps`.

[Vel99] VELDHUIZEN, TODD L.: *Blitz++ User's Guide*, 1999. `http://oonumerics.org/blitz/manual/blitz.ps`.

[VJ97] VELDHUIZEN, TODD L. und M. E. JERNIGAN: *Will C++ be faster than Fortran?* In: *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997. `http://extreme.indiana.edu/~tveldhui/papers/iscope97.ps`.

[Wer92] WERNER, JOCHEN: *Numerische Mathematik 1*. Friedr. Vieweg & Sohn, Braunschweig - Wiesbaden, 1992.

[Wer98] Werner, Jochen: *Numerische Lineare Algebra.* Georg-August-Universität Göttingen, Institut für Numerische und Angewandte Mathematik, Vorlesungsskript, Sommersemester 1998.

[Xie95] Xie, Dexuan: *New parallel iterative methods, new nonlinear multigrid analysis, and application in computational chemistry.* Ph.D. Thesis, University of Houston, 1995. Research report UH/MD 208.
`http://monod.biomath.nyu.edu/~xie/deposit/thesis.ps`.

[You71] Young, David M.: *Iterative Solution of large linear systems.* Academic Press, New York - London, 1971.

Danksagung

Zum Schluß möchte ich allen danken, die mich bei meiner Arbeit unterstützt haben.

Ich danke Gott, der mich nie im Stich gelassen hat.
Besonderer Dank gilt Herrn Prof. Dr. Gert Lube, der stets Zeit für mich hatte und mich intensiv betreut und motiviert hat. Ich danke Andreas Priesnitz für seine vielen Anregungen, die langen Diskussionen und für das unermüdliche Korrekturlesen, für das ich mich außerdem bei Susi, Aniela und Tobias zu bedanken habe. Darüber hinaus bedanke ich mich bei Gert für seine Anregungen und Kommentare. Nicht zuletzt gilt mein Dank auch Herrn Könnecke und Herrn Waßmann vom Operating, die mir bei technischen Problemen zur Seite standen, sowie Herrn Siebrasse und Herrn Weihofen für die u51.

Aus tiefstem Herzen danke ich meinen Eltern, die mir mein Studium ermöglicht und mich ermutigt haben. Dem Wohnheim und seinen Bewohnern danke ich für das soziale Netzwerk und die vielen Ablenkungen angenehmer Art.

Göttingen, den 2.1.2001 Nils Klimanis