

Fakultät für Mathematik

Universität Regensburg

Ein Vergleich des Verfahrens der konjugierten Gradienten und Mehrgittermethoden, angewandt auf die diskretisierte Poisson-Gleichung

Bachelor-Arbeit

Michael Bauer

Matrikel-Nummer 1528558

Erstprüfer Prof. Garcke

Zweitprüfer Prof. ...

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VI
Symbolverzeichnis	VII
1 Einleitung	1
2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2	2
2.1 Definition (Poisson-Gleichung)	2
2.2 Bemerkungen	3
2.3 Finite Differenzen-Methode für die Poisson-Gleichung	3
2.3.1 (Zentraler) Differenzenquotient zweiter Ordnung	3
2.3.2 Diskretisierung von Ω	4
2.4 Eigenwerte und Eigenvektoren der Matrix \mathbf{A}	7
2.4.1 Eigenvektoren von \mathbf{A}	7
2.4.2 Eigenwerte von \mathbf{A}	7
2.4.3 Definition (Kondition einer symmetrischen Matrix)	7
2.4.4 Lemma (Kondition der 2D Poisson Matrix)	8
2.4.4.1 Beweis:	8
3 Iterative Lösungsverfahren für lineare Gleichungssysteme	9
3.1 Das Jacobi-Verfahren (Gesamtschrittverfahren)	9
3.1.1 Das allgemeine Jacobi-Iterationsverfahren	10
3.1.2 Das Jacobi-Iterationsverfahren für die Poisson-Gleichung	10
3.2 Das Gauß-Seidel-Verfahren (Einzelschrittverfahren)	11
3.2.1 Das allgemeine Gauss-Seidel-Iterationsverfahren	11

Inhaltsverzeichnis

3.2.2	Das Gauss-Seidel-Iterationsverfahren für die Poisson-Gleichung . .	12
3.3	Warum Einzelschritt- und Gesamtschrittverfahren?	13
3.4	Die SOR-Methode (SOR = successive overrelaxation)	13
3.4.1	Algorithmus des SOR-Verfahrens	13
3.4.2	Satz (Optimaler Parameter ω)	14
3.4.3	SOR in Matrix-Darstellung	14
3.4.4	Satz	14
3.5	Das Verfahren der konjugierten Gradienten	15
3.5.1	Definition (A-orthogonal)	15
3.5.1.1	Bemerkung:	15
3.5.2	Lemma - (A-orthogonaler) Projektionssatz	16
3.5.3	Allgemeiner Algorithmus der konjugierten Gradienten	16
3.5.3.1	Erklärung zum CG-Algorithmus	17
3.5.4	Numerischer Algorithmus der konjugierten Gradienten	18
3.5.5	Satz (Konvergenz des CG-Algorithmus)	18
3.6	Vorkonditioniertes Verfahren der konjugierten Gradienten (PCG)	19
3.6.1	Satz	19
3.6.1.1	Beweis:	19
3.6.2	Der Algorithmus des vorkonditionierten konjugierten Gradienten Verfahrens	20
3.6.3	Die unvollständige Cholesky-Zerlegung	20
3.6.3.1	Definition (Das Muster E)	21
3.6.3.2	Eigenschaften der Matrix \tilde{L}	21
3.6.3.3	Der numerische Algorithmus der unvollständigen Choles- ky Zerlegung	22
3.6.3.4	Bemerkungen:	22
3.6.4	Die modifizierte unvollständige Cholesky-Zerlegung	22
3.6.4.1	Eigenschaften der Matrix \tilde{L}	22
3.6.4.2	Der numerische Algorithmus der modifizierten unvollstän- digen Cholesky-Zerlegung	23
4	Mehrgitterverfahren	24
4.1	Grundideen	24
4.1.1	Beweis der Residuungleichung	24
4.2	Der Zweigitter-Algorithmus	25

Inhaltsverzeichnis

4.3	Der Mehrgitter-Algorithmus	26
5	Ein Vergleich zwischen Iterativen- und Mehrgitter-Methoden	28
5.1	Beispiel einer Poisson Gleichung	28
5.2	Jacobi-Verfahren angewandt auf das Beispiel	29
5.3	Gauss-Seidel-Verfahren angewandt auf das Beispiel	29
5.4	SOR-Verfahren angewandt auf das Beispiel	29
5.5	CG-Verfahren angewandt auf das Beispiel	29
5.6	PCG-Verfahren angewandt auf das Beispiel	29
5.7	Das Mehrgitterverfahren angewandt auf das Beispiel	29
5.7.1	V-Zyklus	29
5.7.2	W-Zyklus	29
A	Anhang	31
A.1	Quelltexte	31

Abbildungsverzeichnis

Tabellenverzeichnis

5.1	eine sinnlose Tabelle	29
5.2	eine kompliziertere Tabelle	29

Symbolverzeichnis

Allgemeine Symbole

Symbol	Bedeutung
a	der Skalar a
\vec{x}	der Vektor \vec{x}
\mathbf{A}	die Matrix \mathbf{A}

1 Einleitung

Viele Prozesse in den Naturwissenschaften, wie Biologie, Chemie und Physik, aber auch der Medizin, Technik und Wirtschaft lassen sich auf partielle Differentialgleichungen (PDG) zurückführen. Das Lösen solcher Gleichungen ist allerdings nicht immer möglich, oder aufwendig.

Eine PDG, die vor Allem in der Physik häufige Verwendung findet, ist die Poisson-Gleichung – eine elliptische partielle Differentialgleichung zweiter Ordnung. So genügt diese Gleichung beispielsweise dem elektrostatischen Potential u zu gegebener Ladungsdichte f , aber auch dem Gravitationspotential u zu gegebener Massendichte f .

Methoden aus der numerischen Mathematik ermöglichen uns nun das Lösen von partiellen Differentialgleichungen mittels computerbasierten Algorithmen. Hierbei wird jedoch nicht die Lösung direkt bestimmt, sondern versucht eine exakte Approximation der Lösung zu erhalten. Dabei ist es wichtig, dass der zugrunde liegende Algorithmus effizient ist.

Um nun die Lösung einer partiellen Differentialgleichung mittels effizienten Algorithmus bestimmen zu können, müssen wir uns im Vorfeld Gedanken darüber machen, wie wir diese am besten erhalten. Eine der zentralen Methoden der Numerik sind Finite Differenzen. Hierbei diskretisiert man das Gebiet, auf dem die PDG definiert ist und führt die Gleichung auf ein lineares Gleichungssystem zurück.

Eine Möglichkeit ein solches lineares Gleichungssystem zu lösen sind iterative Verfahren. Somit hat man ein großartiges Werkzeug, dass eine Lösung innerhalb weniger Iterationsschritte berechnet und auch mit sehr großen Systemen gut zurecht kommt. Natürlich gilt das nicht für jedes Verfahren, weshalb wir über die Verfahren sprechen möchten, die diese Kriterien erfüllen.

Abschließend wollen wir uns dann noch mit Mehrgittermethoden beschäftigen. Sie stellen die wohl effizienteste und modernste Methode dar, ein lineares Gleichungssystem zu lösen.

2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2

Um die Poisson-Gleichung zu diskretisieren, werden wir diese zunächst definieren. Außerdem werden wir die Methode der finiten Differenzen einführen, um dann ein Gleichungssystem der Form $\mathbf{A}u = f$ zu erhalten.

2.1 Definition (Poisson-Gleichung)

Sei $\Omega = (0,1) \times (0,1) \in \mathbb{R}^2$ ein beschränktes, offenes Gebiet des \mathbb{R}^2 . Gesucht wird eine Funktion $u(x,y)$, die das Randwertproblem

$$-\Delta u(x,y) = f(x,y) \text{ in } \Omega \quad (2.1)$$

$$u(x,y) = g(x,y) \text{ in } \partial\Omega \quad (2.2)$$

löst. Dabei seien $f : \Omega \rightarrow \mathbb{R}$ und $g : \partial\Omega \rightarrow \mathbb{R}$ stetige Funktionen und es bezeichnet $\Delta := \sum_{k=1}^n \frac{\partial^2}{\partial x_k^2}$ den Laplace-Operator. Für die Poisson-Gleichung im \mathbb{R}^2 gilt dann:

$$-\Delta u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \text{ in } \Omega \quad (2.3)$$

$$u(x,y) = g(x,y) \text{ in } \partial\Omega \quad (2.4)$$

(2.2) und (2.4) nennt man Dirichlet-Randbedingung.

2.2 Bemerkungen

- Die Funktion $u(x, y)$ ist häufig formelmäßig nicht darstellbar und wird mit Hilfe von numerischen Verfahren in Ω genähert (Parallele numerische Verfahren/Seite 18 unten)
- Man kann zeigen, dass, wenn $\partial\Omega$ aus glatten Liniensegmenten (z.B. Geraden) zusammengesetzt ist und $f(x, y) \in C^1(\Omega), g \in C(\partial\Omega)$ gilt, die Gleichungen (2.1),(2.2) bzw. (2.3),(2.4) eine eindeutige Lösung besitzen (Dahmen, Reusken Seite 463).

Um diese (elliptische) partielle Differentialgleichung nun in Ω zu diskretisieren, bedarf es der Hilfe der Finiten Differenzen Methode.

2.3 Finite Differenzen-Methode für die Poisson-Gleichung

2.3.1 (Zentraler) Differenzenquotient zweiter Ordnung

Wir betrachten ein $(x, y) \in \Omega$ beliebig. Dann gilt für $h > 0$ mit der Taylorformel

$$u(x+h, y) = u(x, y) + h\partial_x u(x, y) + \frac{h^2}{2!}\partial_{xx}u(x, y) + \frac{h^3}{3!}\partial_{xxx}u(x, y) + \dots \quad (2.5)$$

$$u(x-h, y) = u(x, y) - h\partial_x u(x, y) + \frac{h^2}{2!}\partial_{xx}u(x, y) - \frac{h^3}{3!}\partial_{xxx}u(x, y) + \dots \quad (2.6)$$

Analog können wir diese Betrachtung für $u(x, y+h)$ und $u(x, y-h)$ machen.

Löst man nun (2.5) und (2.6) jeweils nach $\partial_{xx}u(x, y)$ auf und addiert die zwei Gleichungen, so erhält man:

$$\partial_{xx}u(x, y) + O(h^2) = \frac{u(x-h, y) - 2u(x, y) + u(x+h, y)}{h^2} \quad (2.7)$$

Ebenso lösen wir nach $\partial_{yy}u(x, y)$ auf und erhalten analog:

$$\partial_{yy}u(x, y) + O(h^2) = \frac{u(x, y-h) - 2u(x, y) + u(x, y+h)}{h^2} \quad (2.8)$$

2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2

Wobei hier $\partial_{xx}u(x, y) = \frac{\partial^2 u(x, y)}{\partial x^2}$ und $\partial_{yy}u(x, y) = \frac{\partial^2 u(x, y)}{\partial y^2}$ gemeint ist. Diese Näherungen nennt man auch (zentralen) Differenzenquotienten der zweiten Ableitung. $O(h^2)$ ist ein Term zweiter Ordnung und wird vernachlässigt.

Somit erhalten wir für $\Delta u(x, y)$ die Näherung

$$\begin{aligned}\Delta u(x, y) &= \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = \partial_{xx}u(x, y) + \partial_{yy}u(x, y) \\ &\approx \frac{u(x-h, y) + u(x+h, y) - 4u(x, y) + u(x, y-h) + u(x, y+h)}{h^2}\end{aligned}\quad (2.9)$$

2.3.2 Diskretisierung von Ω

Mit einem zweidimensionalen Gitter, der Gitterweite h , wobei $h \in \mathbb{Q}$ mit $h = \frac{1}{n}$ und $n \in \mathbb{N}_{>1}$, wird nun das Gebiet Ω diskretisiert. Die Zahl $(n-1)$ gibt uns an, wie viele Gitterpunkte es jeweils in x- bzw. y-Richtung gibt.

Für $i = 1, \dots, (n-1)$ und $j = 1, \dots, (n-1)$ kann man dann $u(x, y)$ auch in der Form $u(x_i, y_j)$ schreiben:

$$u(x_i, y_j) := u(ih, jh) \quad (2.10)$$

und Ω fassen wir als Ω_h auf, so dass:

$$\Omega_h := \{u(ih, jh) | 1 \leq i, j \leq (n-1)\} \quad (2.11)$$

Betrachten wir nun noch den Rand von Ω_h :

$$\overline{\Omega}_h := \{u(ih, jh) | 0 \leq i, j \leq n\} \quad (2.12)$$

2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2

Mit der Formel (2.9) ergibt sich nun für $\Delta u(x, y)$ die diskretisierte Form:

$$\begin{aligned}
 \Delta_h u(x, y) &:= \frac{u(x-h, y) + u(x+h, y) - 4u(x, y) + u(x, y-h) + u(x, y+h)}{h^2} \\
 &= \frac{u(x_i-h, y_i) + u(x_i+h, y_i) - 4u(x_i, y_i) + u(x_i, y_i-h) + u(x_i, y_i+h)}{h^2} \\
 &= \frac{u(ih-h, jh) + u(ih+h, jh) - 4u(ih, jh) + u(ih, jh-h) + u(ih, jh+h)}{h^2} \\
 &= \frac{1}{h^2} \begin{pmatrix} \frac{u(x, y+h)}{u(x, y)}, & 1, & \frac{u(x, y-h)}{u(x, y)} \end{pmatrix} \begin{pmatrix} 1 & & \\ & -4 & \\ & & 1 \end{pmatrix} \begin{pmatrix} u(x+h, y) \\ u(x, y) \\ u(x-h, y) \end{pmatrix} \quad (2.13)
 \end{aligned}$$

Diese Approximation wird auch 5-Punkt-Differenzenstern genannt, siehe dazu Abbildung (2.1).

Nummeriert man nun alle Gitterpunkte des Gitters fortlaufend von links unten nach rechts oben durch (??) und stellt für jeden dieser Punkte die Gleichung (2.13) auf, so führt dies auf eine $(n-1)^2 \times (n-1)^2$ -Matrix der Form:

$$\mathbf{A} = \frac{1}{h^2} \begin{pmatrix} A_1 & -Id & & \\ -Id & A_2 & \ddots & \\ & \ddots & \ddots & -Id \\ & & -Id & A_n \end{pmatrix} \quad (2.14)$$

wobei $\mathbf{Id} \in \mathbb{R}^{(n-1) \times (n-1)}$ die Identität meint und für alle $i = 0, \dots, n$ gilt:

$$A_i = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \quad (2.15)$$

mit $A_i \in \mathbb{R}^{(n-1) \times (n-1)}$ symmetrisch, positiv definit (s.p.d.).

Um nun auf ein lineares Gleichungssystem der Form $\mathbf{A}u = f$ zu kommen, muss natürlich noch die rechte Seite, also das f aufgestellt werden. Aus der Form der Matrix ist erkennbar, dass nicht alle Punkt aus $\overline{\Omega}$ in \mathbf{A} auftauchen. Das liegt daran, dass die Randpunkte die aus der Dirichlet-Randbedingung resultieren ($u(x, y) = f(x, y)$) bereits bekannt sind und

2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2

somit nicht genähert werden müssen. Jedoch muss zu jeder Komponente in f , die einen Randpunkt als Nachbarn hat, dieser aufaddiert werden. Dies führt uns auf folgende rechte Seite:

$$f = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{pmatrix} \quad (2.16)$$

wobei gilt

$$f_1 = \begin{pmatrix} f(h, h) + h^{-2}(g(h, 0) + g(0, h)) \\ f(2h, h) + h^{-2}(g(2h, 0)) \\ \vdots \\ f(1 - 2h, h) + h^{-2}(g(1 - 2h, 0)) \\ f(1 - h, h) + h^{-2}(g(1 - h, 0) + g(0, 1 - h)) \end{pmatrix} \quad (2.17)$$

$$f_j = \begin{pmatrix} f(h, jh) + h^{-2}(g(0, jh)) \\ f(2h, jh) \\ \vdots \\ f(1 - 2h, jh) \\ f(1 - h, jh) + h^{-2}(g(1, jh)) \end{pmatrix} \quad 2 \leq j \leq n - 2, \quad (2.18)$$

$$f_{n-1} = \begin{pmatrix} f(h, 1 - h) + h^{-2}(g(h, 1) + g(0, 1 - h)) \\ f(2h, 1 - h) + h^{-2}(g(2h, 1)) \\ \vdots \\ f(1 - 2h, 1 - h) + h^{-2}(g(1 - 2h, 1)) \\ f(1 - h, 1 - h) + h^{-2}(g(1 - h, 1) + g(1, 1 - h)) \end{pmatrix} \quad (2.19)$$

Nun steht das lineare Gleichungssystem der Form $\mathbf{A}u = f$, wobei \mathbf{A} und f bekannt sind und u der Lösungsvektor ist, der die Lösung der partielle Differentialgleichung enthält.

2.4 Eigenwerte und Eigenvektoren der Matrix \mathbf{A}

Die Kondition der Matrix \mathbf{A} spielt in den iterativen Verfahren, wie auch bei den Mehrgittermethoden eine große Rolle. Sie entscheidet über die Konvergenzeigenschaften und die Effizienz des Algorithmus. Darum wollen wir uns in diesem Kapitel mit den Eigenwerten und Eigenvektoren der Matrix \mathbf{A} genauer befassen.

Da \mathbf{A} symmetrisch ist, existiert eine Orthogonalbasis aus Eigenvektoren für die gilt:

$$\mathbf{A}x = \lambda x \quad (2.20)$$

2.4.1 Eigenvektoren von \mathbf{A}

Die Eigenvektoren für die Matrix \mathbf{A} aus Gleichung 2.14 lassen sich in folgender Form darstellen:

$$u_{ij}(x, y) = \sin(i\pi x) \sin(j\pi y) \text{ für } i, j \in \mathbb{N} \text{ und } (x, y) \in \Omega_h \quad (2.21)$$

2.4.2 Eigenwerte von \mathbf{A}

Die zugehörigen Eigenwerte lassen sich wie folgt darstellen:

$$\lambda_{ij} = \frac{4}{h^2} \left(\sin^2\left(\frac{i\pi h}{2}\right) + \sin^2\left(\frac{j\pi h}{2}\right) \right) \text{ für } 1 \leq i, j \leq n-1 \quad (2.22)$$

wobei $(n-1)$ die Anzahl der Gitterpunkte in x- und in y-Richtung definiert (siehe oben).

Für das Lemma Unterabschnitt 2.4.4 ist folgende Definition hilfreich:

2.4.3 Definition (Kondition einer symmetrischen Matrix)

Sei \mathbf{A} eine symmetrische Matrix des $\mathbb{R}^{n \times n}$. Dann gilt für die euklidische Norm:

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}} \quad (2.23)$$

2 Diskretisierung der Poisson-Gleichung im \mathbb{R}^2

2.4.4 Lemma (Kondition der 2D Poisson Matrix)

Für die Matrix A der diskreten 2D Poisson Gleichung gilt bzgl. der euklidischen Norm:

$$\kappa_2(A) = \frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} = \left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)) \quad (2.24)$$

2.4.4.1 Beweis:

Da A s.p.d. ist, gilt mit Gleichung 2.23 $\kappa_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$. Für die Eigenwerte gilt aus Gleichung 2.22:

$$\lambda_{ij} = \frac{4}{h^2} \left(\sin^2\left(\frac{i\pi h}{2}\right) + \sin^2\left(\frac{j\pi h}{2}\right) \right) \text{ für } 1 \leq i, j \leq n-1 \quad (2.25)$$

Dann folgt:

$$\kappa_2(A) = \frac{\lambda_{n-1,n-1}}{\lambda_{1,1}} = \frac{\sin^2(\frac{\pi}{2} - \frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} = \frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} \quad (2.26)$$

Substituiert man nun $\cos^2(\frac{\pi h}{2}) = 1 - \sin^2(\frac{\pi h}{2})$ (Pythagoras) und nähert $\sin(\frac{\pi h}{2}) = \frac{\pi h}{2} + \mathcal{O}(h^3)$ mit der Taylorformel, so erhält man:

$$\frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} = \left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)) \quad (2.27)$$

Natürlich wollen wir die Funktion $u(x, y)$ so gut wie möglich in Ω_h approximieren und sind daher bestrebt das Gitter so fein als möglich zu wählen. Daraus ergibt sich jedoch die negative Eigenschaft der 2D Poisson Matrix.

Zur Erinnerung: $h := \frac{1}{n}$. Je größer n gewählt wird, desto schlechter wird die Kondition der Matrix.

$$\lim_{n \rightarrow \infty} \left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)) = \lim_{n \rightarrow \infty} \left(\frac{2}{\pi \frac{1}{n}}\right)^2 (1 + \mathcal{O}(\frac{1}{n^2})) \approx \lim_{n \rightarrow \infty} 8n^2 \rightarrow \infty \quad (2.28)$$

Diese Problematik wird uns in späteren Abschnitten nochmals begegnen.

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

Gleichungssysteme, die partielle Differentialgleichungen lösen können sehr groß werden, da man das entsprechende Gitter sehr fein wählen will, um eine möglichst genaue Lösung zu erhalten. Aus diesem Grund sind direkte Verfahren, wie z.B. der Gauß-Algorithmus oder die LR-Zerlegung nicht geeignet. Ihr Rechenaufwand beläuft sich im Allgemeinen auf $\mathcal{O}(n^3)$ und ist dadurch zu langsam. Fehler - durch Maschinengenauigkeit bedingt - werden außerdem in diesen Verfahren verstärkt. Darum müssen bessere Verfahren gewählt werden.

Ein wesentlicher Bestandteil der numerischen Mathematik sind iterative Verfahren zur Lösung linearer Gleichungssysteme. Diese zeichnen sich meist durch eine schnelle Konvergenz und einen geringen Rechenaufwand aus. Wir wollen uns im folgenden dem Jacobi-Verfahren (Gesamtschrittverfahren), Gauß-Seidel-Verfahren (Einzelschrittverfahren), SOR-Verfahren (SOR = successive overrelaxation) und dem CG-Verfahren (ohne und mit Vorkonditionierung) widmen.

3.1 Das Jacobi-Verfahren (Gesamtschrittverfahren)

Im folgenden betrachten wir das oben beschriebene Gitter mit $(N - 1)$ Gitterpunkten in x- und y-Richtung und erhalten dadurch für die Dimension $n = (N - 1) \cdot (N - 1)$.

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

3.1.1 Das allgemeine Jacobi-Iterationsverfahren

Sei $A \in \mathbb{R}^{n \times n}$ und $f, u \in \mathbb{R}^n$, wobei u die Lösung des linearen Gleichungssystems $Au = f$ ist. Dann lässt sich A wie folgt zerlegen:

$$A = D - L - U \quad (3.1)$$

Dabei sind $D, L, U \in \mathbb{R}^{n \times n}$, wobei D die Diagonalelemente von A enthält, L eine strikte untere und U eine strikte obere Dreiecksmatrix sind.

Somit ergibt sich für $Au = f$:

$$Au = f \Leftrightarrow (D - L - U)u = f \Leftrightarrow Du = (L + U)u + f \quad (3.2)$$

Ist nun D nicht singulär, so gilt für das Jacobi-Verfahren folgende Iterationsvorschrift:

$$Du^{k+1} = (L + U)u^k + f \Leftrightarrow u^{k+1} = D^{-1}(L + U)u^k + D^{-1}f \quad (3.3)$$

Mit der Iterationsmatrix $T := D^{-1}(L + U)$. Wobei dies in Komponentenschreibweise wie folgt aussieht:

Mit einem Startvektor $u^0 \in \mathbb{R}^n$ und $k = 1, 2, \dots$ berechne für $i = 1, \dots, n$:

$$u_i^k = \frac{1}{a_{ii}} \left(f_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} u_j^{k-1} \right) \quad (3.4)$$

Offensichtlich wird jedes u^k durch seinen Vorgänger berechnet. Der Rechenaufwand pro Iterationsschritt beträgt hier $\mathcal{O}(n^2)$ und entspricht somit einer Matrix-Vektor-Multiplikation.

3.1.2 Das Jacobi-Iterationsverfahren für die Poisson-Gleichung

Da die Matrix, die durch das diskretisierte Poisson-Problem aufgestellt wird, dünn besetzt ist, können wir den Rechenaufwand für dieses spezielle Problem auf $\mathcal{O}(n)$ pro Iterationsschritt verbessern. Dafür benötigen wir nochmals den 5-Punkt-Differenzenstern und die partielle Differentialgleichung:

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

$$\frac{u(x_{i-1}, y_j) + u(x_{i+1}, y_j) - 4u(x_i, y_j) + u(x_i, y_{j-1}) + u(x_i, y_{j+1})}{h^2} = f(x, y) \quad (3.5)$$

Löst man diese Gleichung nun nach $u(x_i, y_j)$ auf und führt dies für alle $u(x_i, y_i)$ durch, so erhält man folgende Iterationsvorschrift für das Jacobi-Verfahren:

Für $k = 1, 2, \dots$ berechne mit Startvektor $u^0 \in \mathbb{R}^n$

Für $i = 1, \dots, N - 1$

hspace1cm Für $j = 1, \dots, N - 1$

$$u_{i,j}^k = \frac{1}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k - h^2 f(x_i, y_j)) \quad (3.6)$$

Der Rechenaufwand beträgt nun pro Iterationsschritt lediglich $\mathcal{O}((N - 1) \cdot (N - 1)) = \mathcal{O}(n)$ Schritte.

Ein weiteres Verfahren, welches für die diskretisierte Poisson-Gleichung sogar doppelt so schnell konvergiert, als das Jacobi-Verfahren wollen wir nun im nächsten Abschnitt betrachten.

3.2 Das Gauß-Seidel-Verfahren (Einzelschrittverfahren)

3.2.1 Das allgemeine Gauss-Seidel-Iterationsverfahren

Mit der selben Vorschrift und den selben Überlegungen wie oben, wollen wir die Matrix \mathbf{A} wie folgt zerlegen:

$$A = D - L - U \quad (3.7)$$

Somit ergibt sich für $\mathbf{A}u = f$:

$$Au = f \Leftrightarrow (D - L - U)u = f \Leftrightarrow (D - L)u = Uu + f \quad (3.8)$$

Daraus können wir nun folgende Iterationsvorschrift ableiten:

$$(D - L)u^{k+1} = Uu^k + f \Leftrightarrow u^{k+1} = (D - L)^{-1}Uu^k + (D - L)^{-1}f \quad (3.9)$$

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

Dies nun in Komponentenschreibweise für $i = 1, \dots, n$:

$$\sum_{j=1}^i a_{ij} u_j^{k+1} = - \sum_{j=i+1}^n a_{ij} u_j^k + f_i \quad (3.10)$$

Formt man Gleichung 3.10 um, so erhält man den Algorithmus des Gauss-Seidel-Verfahrens mit Startvektor $u^0 \in \mathbb{R}$:

Für $k = 1, 2, \dots$ berechne für $i = 1, \dots, n$

$$u_i^{k+1} = \frac{1}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} u_j^{k+1} - \sum_{j=i+1}^n a_{ij} u_j^k \right) \quad (3.11)$$

Wie zu sehen ist, verwendet dieser Algorithmus dieses mal nicht nur die Werte aus dem vorigen Iterationsschritt, sondern auch welche aus dem gerade berechnet wird. Das Gauss-Seidel-Verfahren wartet ebenfalls mit einem Rechenaufwand von $\mathcal{O}(n^2)$ auf. Wie beim Jacobi-Verfahren kann man dies auf $\mathcal{O}(n)$ optimieren.

3.2.2 Das Gauss-Seidel-Iterationsverfahren für die Poisson-Gleichung

Um nun eine angepasste Formel bzw. Iterationsvorschrift zu erhalten gehen wir wieder mit Hilfe des 5-Punkt-Differenzensterns vor. Allerdings verwendet wie oben beschrieben der Algorithmus auch Werte, die im aktuellen Iterationsschritt berechnet wurden. Um dies zu gewährleisten müssen wir uns nur ?? ansehen. Hierbei ist ersichtlich, dass $u_{i-1,j}$ und $u_{i,j-1}$ bereits neu berechnet wurden. $u_{i+1,j}$ und $u_{i,j+1}$ stammen noch aus dem vorigen Iterationsschritt.

Daraus ergibt sich der spezielle Gauss-Seidel-Algorithmus für die Poisson-Gleichung:

$$u_{i,j}^k = \frac{1}{4} (u_{i-1,j}^k + u_{i,j-1}^k + u_{i+1,j}^{k-1} + u_{i,j+1}^{k-1} - h^2 f(x_i, y_j)) \quad (3.12)$$

Auch hier reduziert sich der Rechenaufwand auf $\mathcal{O}(n)$.

3.3 Warum Einzelschritt- und Gesamtschrittverfahren?

Zum lösen von großen linearen Gleichungssystemen werden das Jacobi-Verfahren und das Gauss-Seidel-Verfahren heute kaum mehr verwendet. Es gibt mittlerweile schnellere, stabilere und effizientere Verfahren wie wir im fortlaufenden sehen werden. Einen großen Vorteil haben allerdings beide Verfahren: Sie löschen große Fehler bereits nach den ersten Iterationsschritten aus. Darum finden sie besonders große Verwendung in den Mehrgittermethoden, die wir später kennen lernen werden.

3.4 Die SOR-Methode (SOR = successive overrelaxation)

Eine Optimierung von Gleichung 3.10 bietet das SOR-Verfahren oder auch Gauss-Seidel-Relaxationsverfahren. Die Einführung eines Relaxationsparameters ω kann in einigen Fällen das Iterationsverfahren effizienter machen. Speziell für die Poisson-Gleichung wird nicht nur die Effizienz signifikant verbessert, es ist auch ein optimaler Parameter ω bekannt (siehe hierzu Dahmen/Reusken).

3.4.1 Algorithmus des SOR-Verfahrens

Für $k = 1, 2, \dots$ berechne für $i = 1, \dots, n$

$$u_i^{k+1} = u_i^k - \frac{\omega}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} u_j^{k+1} - \sum_{j=i}^n a_{ij} u_j^k \right) \quad (3.13)$$

$$= (1 - \omega) u_i^k - \frac{\omega}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} u_j^{k+1} - \sum_{j=i+1}^n a_{ij} u_j^k \right) \quad (3.14)$$

Offensichtlich erhält man für $\omega = 1$ wieder das Gauss-Seidel-Verfahren. Außerdem existiert, wie bereits erwähnt, nicht immer ein ω , welches das Verfahren schneller konvergieren lässt. Da für die Poisson Gleichung sogar ein optimaler Parameter existiert, wollen wir uns diesen näher ansehen.

3.4.2 Satz (Optimaler Parameter ω)

Sei $\mu := \cos(\pi h)$. Für die diskretisierte Poisson Gleichung $Au = f$ gilt dann für den Relaxationsparameter:

$$\omega_{opt} := 1 + \left(\frac{\mu}{1 + \sqrt{1 - \mu^2}} \right) \quad (3.15)$$

Vergleichen Sie hierzu auch Dahmen/Reusken (Seite 564).

3.4.3 SOR in Matrix-Darstellung

Der Vollständigkeit halber, wollen wir uns jetzt noch der Matrix-Darstellung der SOR-Methode widmen. Im wesentlichen entspricht sie dem Gauss-Seidel-Verfahren. Lediglich der Relaxationsparameter muss eingebaut werden. Man erhält:

$$\begin{aligned} (\mathbf{Id} + \omega \mathbf{D}^{-1} \mathbf{L}) x^{k+1} &= [(1 - \omega) \mathbf{I} - \omega \mathbf{D}^{-1} \mathbf{R}] x^k + \omega \mathbf{D}^{-1} b \\ \Leftrightarrow \mathbf{D}^{-1} (\mathbf{D} + \omega \mathbf{L}) x^{k+1} &= \mathbf{D}^{-1} [(1 - \omega) \mathbf{D} - \omega \mathbf{R}] x^k + \omega \mathbf{D}^{-1} b \end{aligned} \quad (3.16)$$

dies ergibt

$$x^{k+1} = (\mathbf{D} + \omega \mathbf{L})^{-1} [(1 - \omega) \mathbf{D} - \omega \mathbf{R}] x^k + \omega (\mathbf{D} + \omega \mathbf{L})^{-1} b \quad (3.17)$$

Mit dieser Gleichung und einigen weiteren Überlegungen kann gezeigt werden, dass folgender Satz gilt.

3.4.4 Satz

Sei \mathbf{A} hermitesch und positiv definit, dann konvergiert das Gauß-Seidel-Relaxationsverfahren genau dann, wenn $\omega \in (0, 2)$ ist. (aus oranges buch)

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

Einen Beweis hierzu findet man beispielsweise in (oranges buch)

Aus Unterabschnitt 3.4.2 ist der optimale Parameter für die Poisson Gleichung bereits bekannt. Also konvergiert - wie erwartet - das Verfahren für $\omega =$.

3.5 Das Verfahren der konjugierten Gradienten

Das Verfahren der konjugierten Gradienten wurde 1952 von Heestens und Stiefel erstmals vorgestellt. Das Verfahren zeichnet sich durch Stabilität und schnelle Konvergenz aus. Das Verfahren der konjugierten Gradienten (auch CG-Verfahren) ist eine Krylov-Unterraum-Methode und gehört zu den Projektionsverfahren. Charakteristisch für das CG-Verfahren ist außerdem die A-Orthogonalität der Basen des Krylovraums. Die Orthogonalität hängt also von der zugrunde liegenden Matrix **A** ab. Aus diesem Grund müssen wir zunächst definieren, was A-orthogonal heißt.

3.5.1 Definition (A-orthogonal)

Sei **A** eine symmetrische, nicht singuläre Matrix. Zwei Vektoren $x, y \in \mathbb{R}^n$ heißen konjugiert oder A-orthogonal, wenn $x^T A y = 0$ gilt.

3.5.1.1 Bemerkung:

- Es definiert $\langle x, y \rangle_A = x^T A y$ ein Skalarprodukt auf dem \mathbb{R}^n für **A** s.p.d.
- Wir nennen $\|x\|_A := \sqrt{\langle x, x \rangle_A}$ die Energie-Norm.

Nun wollen wir noch den Projektionssatz in Abhängigkeit unserer Matrix **A** definieren.

3.5.2 Lemma - (A-orthogonaler) Projektionssatz

Sei U_k ein k -dimensionaler Teilraum des \mathbb{R}^n ($k \leq n$), und p^0, p^1, \dots, p^{k-1} eine A -orthogonale Basis dieses Teilraums, also $\langle p^i, p^j \rangle_A = 0$ für $i \neq j$. Sei $v \in \mathbb{R}^n$, dann gilt für $u^k \in U_k$:

$$\|u^k - v\|_A = \min_{u \in U_k} \|u - v\|_A \quad (3.18)$$

genau dann, wenn u^k die A -orthogonale Projektion von v auf $U_k = \text{span}\{p^0, \dots, p^{k-1}\}$ ist. Außerdem hat u^k die Darstellung

$$P_{U_k, \langle \cdot, \cdot \rangle_A}(v) = u^k = \sum_{j=0}^{k-1} \frac{\langle v, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \quad (3.19)$$

Der Beweis zu diesem Lemma folgt direkt aus dem Projektionssatz.

Was dieser Satz nun aussagt ist, dass man einen Vektor v in U_{k+1} sucht. Nun wählen wir ein beliebiges $v \in U_{k+1}$ und minimiert über alle $u^k \in U_k$. Die optimale Lösung ist dann der gesuchte Vektor v . Bildlich gesprochen, ist v die (A-)orthogonale Projektion auf U_k . Vergleiche Bild (bla).

Da wir nun die Grundlagen für das CG-Verfahren geschaffen haben, wollen wir nun den Algorithmus betrachten.

3.5.3 Allgemeiner Algorithmus der konjugierten Gradienten

Zur Erzeugung der Lösung von x^* durch Näherungen x^1, x^2, \dots definieren wir folgende Teilschritte:

0. Definiere den ersten Teilraum und bestimme das (Start-) Residuum mit Startvektor x^0

$$U_1 := \text{span}\{r^0\}, \text{ wobei } r^0 = b - Ax^0 \quad (3.20)$$

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

1. Bestimme eine A-orthogonale Basis

$$p^0, \dots, p^{k-1} \text{ von } U_k \quad (3.21)$$

2. Bestimme eine Näherungslösung x^k , so dass

$$\|x^k - x^*\|_A = \min_{u \in U_k} \|x - x^*\|_A \quad (3.22)$$

gilt. Mit dem A-orthogonalen Projektionssatz berechnen wir also:

$$x^k = \sum_{j=0}^{k-1} \frac{\langle x^*, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \quad (3.23)$$

3. Erweitere den Teilraum U_k und berechne das iterierte Residuum

$$U_{k+1} := \text{span}\{p^0, \dots, p^{k-1}, r^k\} \text{ wobei } r^k := b - Ax^k \quad (3.24)$$

Natürlich ist das (noch) kein numerischer Algorithmus, den man in Programmcode umsetzen kann, allerdings sollte man sich die Schritte des CG-Algorithmus klar machen, um die Effizienz dahinter zu verstehen:

3.5.3.1 Erklärung zum CG-Algorithmus

Vielleicht das Wichtigste vorab: Der Algorithmus endet nach maximal n Schritten. Da wir die Lösung $u \in \mathbb{R}^n$ suchen und unsere Teilräume U_k mit $k \leq n$ sind, muss nach spätestens n Schritten das Verfahren die optimale Lösung im \mathbb{R}^n gefunden haben. Oftmals ist eine gesuchte Näherung in einem Teilraum der Lösung bereits sehr nahe bzw. gleich der Lösung. Dann bricht der Algorithmus vorzeitig ab.

Um nun das Verfahren zu erklären, betrachten wir nochmals alle Teilschritte des Algorithmus:

zu 0. Was der erste Schritt im wesentlichen aussagt ist, dass man den Algorithmus initialisiert und ein Residuum bestimmen muss. Hierfür ist ein Startvektor x^0 beliebig zu wählen (??). Das Residuum wird durch $r^0 := b - Ax^0$ definiert.

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

- zu 1.** Um eine A-orthogonale Basis von den U_k zu bestimmen ist im wesentlichen eine (A-) Orthogonalisierungsverfahren notwendig, welches hier auch angewandt wird.
- zu 2.** Hier wird das in Unterabschnitt 3.5.2 bereits besprochene Verfahren angewandt, um eine neue Näherungslösung in U_k zu bestimmen.
- zu 3.** Hier soll im wesentlichen das Gleiche geschehen in wie in Schritt 0. Lediglich wird nun das k -te Residuum durch $r^k := b - Ax^k$ bestimmt.

All diese Überlegung führen uns nun zu einem numerischen Algorithmus, den man in Programmcode umsetzen kann. Die einzelnen Herleitungen und Beweise sind z.B. in Dahmen/Reusken (Seiten bla) zu finden.

3.5.4 Numerischer Algorithmus der konjugierten Gradienten

Gegeben ist eine symmetrisch positiv definite Matrix $A \in \mathbb{R}^n$. Bestimme die (Näherungs-) Lösung x^* mit Hilfe eines beliebigen Startvektors $x^0 \in \mathbb{R}^n$ zu einer gegebenen rechten Seite $b \in \mathbb{R}^n$. Setze $\beta_{-1} := 0$ und berechne das Residuum $r^0 = b - Ax^0$.

Für $k = 1, 2, \dots$, falls $r^{k-1} \neq 0$ berechne:

$$\begin{aligned} p^{k-1} &= r^{k-1} + \beta_{k-2} p^{k-2}, \text{ wobei } \beta_{k-2} = \frac{\langle r^{k-1}, r^{k-1} \rangle}{\langle r^{k-2}, r^{k-2} \rangle} \text{ mit } (k \geq 2) \\ x^k &= x^{k-1} + \alpha_{k-1} p^{k-1}, \text{ wobei } \alpha_{k-1} = \frac{\langle r^{k-1}, r^{k-1} \rangle}{\langle p^{k-1}, Ap^{k-1} \rangle} \\ r^k &= r^{k-1} - \alpha_{k-1} Ap^{k-1} \end{aligned}$$

Bevor wir nun eine Weiterentwicklung des CG-Verfahrens betrachten, soll an dieser Stelle ein kurzer Satz über die Konvergenz des Verfahrens folgen. Einen Beweis hierzu findet man z.B. in (Dahmen/Reusken 573):

3.5.5 Satz (Konvergenz des CG-Algorithmus)

Sei $A \in \mathbb{R}^{n \times n}$ symmetrisch, positiv definit und seien $u, f, u^*, u^k \in \mathbb{R}^n$, wobei u^* die exakte Lösung des Gleichungssystems $Au = b$ und x^k die approximierte Lösung durch das

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

CG-Verfahren ist. Dann gilt für $k = 1, 2, \dots$:

$$\|x^k - x^*\|_A \leq 2 \left(\frac{\kappa_2(A) - 1}{\kappa_2(A) + 1} \right)^k \|x^0 - x^*\|_A \quad (3.25)$$

Da stets $\frac{\kappa_2(A)-1}{\kappa_2(A)+1} < 1$ gilt, liefert dieser Satz die Konvergenz des Algorithmus. Man sieht also, dass die Konvergenz des Verfahrens von der Kondition der Matrix \mathbf{A} abhängt.

3.6 Vorkonditioniertes Verfahren der konjugierten Gradienten (PCG)

Das PCG-Verfahren (preconditioned conjugate gradient) ist eine optimierte Version des CG-Verfahrens. Wie wir in Unterabschnitt 2.4.4 gesehen haben, ist die Matrix \mathbf{A} schlecht konditioniert. Die Idee ist nun, die bei der Iterations zu Grunde liegende Matrix \mathbf{A} durch eine ähnliche Matrix mit besserer Kondition zu ersetzen, damit sich das Konvergenzverhalten verbessert.

3.6.1 Satz

Sei $\mathbf{W} \in \mathbb{R}^{n \times n}$ s.p.d. dann gilt:

$$\mathbf{A}u = f \iff \mathbf{W}^{-1}\mathbf{A}x = \mathbf{W}^{-1}b \quad (3.26)$$

3.6.1.1 Beweis:

$$\begin{aligned} \mathbf{A}u = f &\iff u = \mathbf{A}^{-1}\mathbf{E}f \iff u = \mathbf{A}^{-1}\mathbf{W}\mathbf{W}^{-1}f \\ &\iff u = (\mathbf{W}^{-1}\mathbf{A})^{-1}f \iff \mathbf{W}^{-1}\mathbf{A}x = \mathbf{W}^{-1}b \end{aligned}$$

Die Konditionszahl dieses Problem ist nun durch $\kappa_2(\mathbf{W}^{-1}\mathbf{A})$ bedingt. Das Ziel muss es also sein, \mathbf{W}^{-1} so gut wie möglich zu wählen, damit die Kondition möglichst klein wird. Nun ist

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

im Allgemeinen $\mathbf{W}^{-1}\mathbf{A}$ nicht s.p.d. Somit könnten wir zwar den CG-Algorithmus trotzdem darauf anwenden, werden aber wegen dieser Tatsache möglicherweise keine Konvergenz erhalten. Um dies zu umgehen findet man z.B. in (Dahmen/Reusken 576) einen Lösungsansatz, bei dem mit der Cholesky-Zerlegung eine entsprechende Umformung gefunden werden kann.

3.6.2 Der Algorithmus des vorkonditionierten konjugierten Gradienten Verfahrens

Gegeben seien $\mathbf{A}, \mathbf{W} \in \mathbb{R}^n$ s.p.d. Bestimme die (Näherungs-) Lösung x^* mit Hilfe eines beliebigen Startvektors $x^0 \in \mathbb{R}^n$ zu einer gegebenen rechten Seite $b \in \mathbb{R}^n$. Setze $\beta_{-1} := 0$, berechne das Residuum $r^0 = b - Ax^0$ und $z^0 = \mathbf{W}^{-1}r^0$ (löse $\mathbf{W}z^0 = r^0$). Für $k = 1, 2, \dots$, falls $r^{k-1} \neq 0$ berechne:

$$\begin{aligned} p^{k-1} &= z^{k-1} + \beta_{k-2} p^{k-2}, \text{ wobei } \beta_{k-2} = \frac{\langle z^{k-1}, r^{k-1} \rangle}{\langle z^{k-2}, r^{k-2} \rangle} \text{ mit } (k \geq 2) \\ x^k &= x^{k-1} + \alpha_{k-1} p^{k-1}, \text{ wobei } \alpha_{k-1} = \frac{\langle z^{k-1}, r^{k-1} \rangle}{\langle p^{k-1}, Ap^{k-1} \rangle} \\ r^k &= r^{k-1} - \alpha_{k-1} Ap^{k-1} \\ z^k &= \mathbf{W}^{-1}r^k \text{ (löse } \mathbf{W}z^k = r^k) \end{aligned}$$

Wichtig hierbei ist, dass das Lösen von $\mathbf{W}z^k = r^k$ mit möglichst wenig Aufwand (ideal: $\mathcal{O}(n)$) berechnet werden soll.

3.6.3 Die unvollständige Cholesky-Zerlegung

Eine Matrix \mathbf{A} , die symmetrisch positiv definit ist, lässt sich durch eine Cholesky-Zerlegung in eine normierte untere Dreiecksmatrix \mathbf{L} und eine rechte obere Dreiecksmatrix \mathbf{U} zerlegen, wobei gilt: $\mathbf{U} := \mathbf{D}\mathbf{L}^T$

Mit dieser Zerlegung möchten wir nun unser System vorkonditionieren. Allerdings würde

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

eine vollständige Cholesky-Zerlegung viele Nulleinträge in einer Sparse-Matrix auslösen. Darum greift man auf eine unvollständige Cholesky-Zerlegung zurück, bei der die Stellen, an denen \mathbf{A} Nulleinträge besitzt, ebenfalls Null werden.

3.6.3.1 Definition (Das Muster E)

Sei das Muster $E \subseteq \{1, \dots, n\} \times \{1, \dots, n\}$, dann gilt:

$$E := \{(i, j) | 1 \leq i, j \leq n, a_{i,j} \neq 0\} \quad (3.27)$$

Dann lässt sich die Matrix \mathbf{A} auch folgendermaßen schreiben:

$$\mathbf{A} = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T + E \approx \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T \quad (3.28)$$

wobei $\tilde{\mathbf{L}}, \tilde{\mathbf{L}}^T$ nicht die komplette Faktorisierung darstellt, sondern folgende Eigenschaften erfüllt:

3.6.3.2 Eigenschaften der Matrix $\tilde{\mathbf{L}}$

- $\tilde{\mathbf{L}}$ ist normierte untere Dreiecksmatrix
- Es gilt: $l_{i,j} = 0$, falls $(i, j) \notin E$

Natürlich ist diese Faktorisierung ungenauer, als die vollständige Zerlegung, allerdings genügt sie, um die Kondition des Gleichungssystems in vielen Fällen zu verbessern. Um den folgenden Algorithmus effizient zu machen, werden Summen nur über Indizes aus dem Muster berechnet.

3.6.3.3 Der numerische Algorithmus der unvollständigen Cholesky Zerlegung

Seien $\mathbf{A} \in \mathbb{R}^{n \times n}$ s.p.d. und E das Muster zur Matrix \mathbf{A} . Berechne dann für $i = 1, 2, \dots, n$:

$$l_{i,i} = \left(a_{i,i} - \sum_{j=1, (i,j) \in E}^{i-1} l_{i,j}^2 \right)^{\frac{1}{2}} \quad (3.29)$$

$$\text{for } k = i + 1, \dots, n : \text{ if } (k, i) \in E : \quad (3.30)$$

$$l_{k,i} = \left(a_{k,i} - \sum_{j=1, (k,j) \in E, (i,j) \in E}^{k-1} l_{k,j} l_{i,j} \right) / l_{i,i} \quad (3.31)$$

3.6.3.4 Bemerkungen:

- Die für den PCG-Algorithmus wichtige Matrix \mathbf{W} wird nun definierte als: $\mathbf{W} := \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$. Dadurch wird auch $\mathbf{W}z^k = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T z^k = r^k$ schnell durch Vorwärts- bzw. Rückwärts-einsetzen lösbar.
- Für viele Probleme zeigt sich, dass $\kappa_2(\mathbf{W}^{-1}\mathbf{A}) \ll \kappa_2(\mathbf{A})$ gilt.

Es gibt einige Varianten dieses Verfahrens. Wir wollen uns an dieser Stelle mit einer auseinander setzen.

3.6.4 Die modifizierte unvollständige Cholesky-Zerlegung

Auch bei der modifizierten Methode des Verfahrens gehen wir vor, wie in Unterabschnitt 3.6.3. Jedoch werden die Vorschriften für die Matrix $\tilde{\mathbf{L}}$ abgeändert:

3.6.4.1 Eigenschaften der Matrix $\tilde{\mathbf{L}}$

Sei $e := (1, 1, \dots, 1)^T$

- $a_{i,j} = (\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T)_{i,j}$ für alle $(i, j) \in E, i \neq j$

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

- $\mathbf{A}e = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T e$, d.h. die Zeilensummen stimmen überein.
- $l_{i,j} = r_{i,j} = 0$ für alle $(i,j) \notin E$

3.6.4.2 Der numerische Algorithmus der modifizierten unvollständigen Cholesky-Zerlegung

Seien $\mathbf{A} \in \mathbb{R}^{n \times n}$ s.p.d. und E das Muster zur Matrix \mathbf{A} . Berechne dann für $i = 1, 2, \dots, n$:

$$l_{i,i} = \left(a_{i,i} - \sum_{j=1, (i,j) \in E}^{i-1} l_{i,j}^2 \right)^{\frac{1}{2}} \quad (3.32)$$

$$\text{for } k = i + 1, \dots, n : \quad (3.33)$$

$$\text{if } (k,i) \in E : \quad (3.34)$$

$$l_{k,i} = \left(a_{k,i} - \sum_{j=1, (k,j) \in E, (i,j) \in E}^{k-1} l_{k,j} l_{i,j} \right) / l_{i,i} \quad (3.35)$$

$$\text{else } : \quad (3.36)$$

$$a_{k,k} = a_{i,i} - \sum_{j=1, (k,j) \in E, (i,j) \in E}^{k-1} l_{k,j} l_{i,j} \quad (3.37)$$

Wir setzen wieder $\mathbf{W} := \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$.

Es gibt noch weitere Verfahren (siehe z.B. Saad), wie beispielsweise das SSOR-Verfahren, die wir hier nicht weiter diskutieren wollen. In Kapitel 5 werden wir die Vorteile der Vorkonditionierung anhand der 2D Poisson Gleichung sehen.

4 Mehrgitterverfahren

In diesem Abschnitt sollen nun die Mehrgittermethoden genauer betrachtet werden. Bevor wir jedoch genauer auf dieses Verfahren eingehen, wollen wir uns nochmal einige Erkenntnisse klar machen:

4.1 Grundideen

1. Auslöschung hochfrequenter Fehler

Das Gauß-Seidel-Verfahren und das Jacobi-Verfahren löschen hochfrequente Fehler in den ersten Iterationsschritten aus. Niederfrequente Fehler werden nur sehr langsam beseitigt. (siehe Abschnitt 3.1 und Abschnitt 3.2)

2. Grobe Fehler nach einer Gittertransformation

Niedrig frequente Fehler auf einem feinen Gitter werden zu hochfrequenten Fehlern, wenn sie auf ein gröberes Gitter überführt werden.

3. Residuums Gleichung

Die für diesen Algorithmus wichtige Residuums Gleichung lautet:

$$\mathbf{A}e^k = r^k \quad (4.1)$$

Die Lösung von $\mathbf{A}e^k = r^k$ ist äquivalent zur Lösung von $\mathbf{A}u = b$, wobei $e^k = 0$.

4.1.1 Beweis der Residuums Gleichung

Das Residuum ist an der k – ten Stelle definiert als

$$r^k = b - \mathbf{A}u^k \quad (4.2)$$

4 Mehrgitterverfahren

Der Fehler

$$e = u^* - u^k \quad (4.3)$$

wobei u^* die exakte Lösung darstellt, erfüllt ebenso folgende Gleichung:

$$\mathbf{A}e^k = \mathbf{A}(u^* - u^k) = \mathbf{A}u^* - \mathbf{A}u^k = b - \mathbf{A}u^k = r^k \quad (4.4)$$

Wir kennen zwar den Fehler e^k nicht, wissen aber, dass dieser 0 ist, falls $r^k = 0$. \implies Beh.

Gauss-Seidel- und Jacobi-Verfahren löschen also hochfrequente Fehler in den ersten Iterationsschritten aus. Um die nieder frequenten Fehler zu reduzieren sind allerdings wesentlich mehr Iterationsschritte notwendig. Auch aus diesem Grund finden beide Verfahren bei der Lösung großer, linearer Gleichungssysteme wenig Anwendung.

Auch wenn die hohe Anzahl an notwendigen Iterationen ein deutlicher Nachteil ist, wollen wir im folgenden die Vorteile dieser Methoden ausnutzen.

Wie in Abschnitt 2.3 gesehen befinden wir uns bei der Diskretisierung der Poisson-Gleichung auf einem Gebiet $\Omega_h = (0,1)^2$ der Schrittweite $h = \frac{1}{n}$. Nach der Ausführung von k -Iterationsschritten von Einzel- oder Gesamtschrittverfahren sind auf diesem Gitter die hochfrequenten Fehler $e^k = u^* - u^k$ verschwunden. Nun berechnet man im k -ten Schritt das Residuum r^k und führt für das äquivalente lineare Gleichungssystem $\mathbf{A}e^k = r^k$, wobei $e^k = 0$ gilt, l Iterationsschritte aus. So erhalten wir eine Näherung des Fehlers e^k .

Stellt man Gleichung 4.3 um, berechnet also $e^k + u^k$, so erhält man eine neue Näherung der exakten Lösung.

Kombiniert man dieses Vorgehen nun mit dem Wechsel zwischen zwei Gittern der Gitterweite h und $2h$ so erhält man das Zweigitterverfahren:

4.2 Der Zweigitter-Algorithmus

Der Zweigitter-Algorithmus findet in der Praxis zwar wenig Verwendung, allerdings wird die Idee dahinter die Basis für das Mehrgitter-Verfahren sein.

4 Mehrgitterverfahren

```
while       $u^k \neq u^*$ 
    pre-smooth    Jacobi-/Gauss-Seidel-Steps
    calculate residual  $r^k = b - Au^k$ 
    restrict       $r_{2h}^k = Rr_h^k$ 
                    $A^{2h} = RA^hP$ 
    set error      $e_{2h}^0 = 0$ 
    solve direct    $A^{2h}e_{2h} = r_{2h}^k$ 
    prolongate     $e_h^k = Pe_{2h}^k$ 
    add error      $u_h^k = u_h^{k-1} + e_h^k$ 
    smooth (optional) Jacobi-/Gauss-Seidel-Steps
end
```

Zunächst sei erwähnt, dass das Nachglätten optional ist, allerdings einige Vorteile bietet, auf die wir hier nicht genauer eingehen möchten. Bei der Implementierung sollte also auf Nachglättung geachtet werden.

Der klare Nachteil dieser Methode liegt natürlich im direkten Lösen der Residuums Gleichung. Wählen wir ein sehr feines Gebiet Ω_h mit $n = 256$, also $h = \frac{1}{256}$, so liefert das Gebiet Ω_{2h} immer noch ein Gleichungssystem der Dimension 127^2 . Ein System dieser Ordnung zu lösen erfordert Rechenaufwand, der hier nicht erwünscht ist.

4.3 Der Mehrgitter-Algorithmus

Eine bessere Methode ist, das Gitter immer feiner zu machen, bis das System direkt lösbar ist, um dann wieder auf das feinste Gitter zurück zu kehren. Genauer:

4 Mehrgitterverfahren

Multigrid (u, b)

```

if (finest grid)    return  $u_{\text{finestgrid}} = \mathbf{A}^{-1}b$ 
    else
        pre-smooth    Jacobi-/Gauss-Seidel-Steps
    calculate residual  $r^k = b - Au^k$ 
        restrict       $r_{2h}^k = Rr_h^k$ 
                         $\mathbf{A}^{2h} = R\mathbf{A}^hP$ 
        recursion      $e_{2h}^k = \mathbf{Multigrid}(0, r_{2h}^k)$ 
        prolongate     $e_h^k = Pe_{2h}^k$ 
        add error      $u_h^k = u_h^{k-1} + e_h^k$ 
        smooth        Jacobi-/Gauss-Seidel-Steps
                        return  $u_h$ 
    end

```

Nun wollen wir die Idee, die wir in Abschnitt 4.2 entwickelt haben formulieren. Hier unterscheiden wir dann zwischen zwei Methoden: dem **V-Zyklus** und dem **W-Zyklus**. Bei einem V-Zyklus wird geht man pro Iterationsschritt vom feinsten zum größten Gitter und zurück (siehe Bild bla). Bei einem W-Zyklus geht man vom feinsten auf das größte Gitter über, prolongiert

5 Ein Vergleich zwischen Iterativen- und Mehrgitter-Methoden

In diesem letzten Kapitel wollen wir die Iterativen-Verfahren, speziell das Verfahren der konjugierten Gradienten, und die Mehrgitterverfahren numerisch vergleichen. Dafür betrachten wir folgende Gleichung:

5.1 Beispiel einer Poisson Gleichung

Seien $f : \Omega \rightarrow \mathbb{R}$ und $g : \partial\Omega \rightarrow \mathbb{R}$ stetige Funktionen mit $f(x, y) = -4$. und $g(x, y) = x^2 + y^2$. Sei außerdem $\Omega = (0, 1) \times (0, 1) \in \mathbb{R}^2$. Gegeben ist das Randwertproblem

$$-\Delta u(x, y) = f(x, y) = -4 \text{ in } \Omega \quad (5.1)$$

$$u(x, y) = g(x, y) = x^2 + y^2 \text{ in } \partial\Omega \quad (5.2)$$

Gesucht ist eine Funktion $u(x, y)$, die diese Gleichung löst.

Offensichtlich löst der elliptische Paraboloid $u(x, y) = x^2 + y^2$ die partielle Differentialgleichung, da $\partial_{xx}u(x, y) = \partial_{yy}u(x, y) = 2$. Allerdings wollen wir nun diese Lösung auch numerisch erhalten.

5.2 Jacobi-Verfahren angewandt auf das Beispiel

5.3 Gauss-Seidel-Verfahren angewandt auf das Beispiel

5.4 SOR-Verfahren angewandt auf das Beispiel

5.5 CG-Verfahren angewandt auf das Beispiel

5.6 PCG-Verfahren angewandt auf das Beispiel

5.7 Das Mehrgitterverfahren angewandt auf das Beispiel

5.7.1 V-Zyklus

5.7.2 W-Zyklus

Formen	Städte
Quadrat	Bunkenstedt
Dreieck	Laggenbeck
Kreis	Peine
Raute	Wakaluba

Tabelle 5.1: eine sinnlose Tabelle

		dies			
		von dort	und dort	über hier	zu Los
das	hier	bla	bla	bla	bla
	dort	bla	bla	bla	bla
	da	bla	bla	bla	bla

Tabelle 5.2: eine kompliziertere Tabelle mit viel Beschreibungstext, der aber nicht im Tabellenverzeichnis auftauchen soll

5 Ein Vergleich zwischen Iterativen- und Mehrgitter-Methoden

Er hörte leise Schritte hinter sich. Das bedeutete nichts Gutes. Wer würde ihm schon folgen, spät in der Nacht und dazu noch in dieser engen Gasse mitten im übel beleumundeten Hafenviertel? Gerade jetzt, wo er das Ding seines Lebens gedreht hatte und mit der Beute verschwinden wollte! Hatte einer seiner zahllosen Kollegen dieselbe Idee gehabt, ihn beobachtet und abgewartet, um ihn nun um die Früchte seiner Arbeit zu erleichtern? Oder gehörten die Schritte hinter ihm zu einem der unzähligen Gesetzeshüter dieser Stadt, und die stählerne Acht um seine Handgelenke würde gleich zuschnappen? Er konnte die Aufforderung stehen zu bleiben schon hören. Gehetzt sah er sich um. Plötzlich erblickte er den schmalen Durchgang. Blitzartig drehte er sich nach rechts und verschwand zwischen den beiden Gebäuden. Beinahe wäre er dabei über den umgestürzten Mülleimer gefallen, der mitten im Weg lag. Er versuchte, sich in der Dunkelheit seinen Weg zu ertasten und erstarrte: Anscheinend gab es keinen anderen Ausweg aus diesem kleinen Hof als den Durchgang, durch den er gekommen war. Die Schritte wurden lauter und lauter, er sah eine dunkle Gestalt um die Ecke biegen. Fieberhaft irrten seine Augen durch die nächtliche Dunkelheit und suchten einen Ausweg. War jetzt wirklich alles vorbei, waren alle Mühe und alle Vorbereitungen umsonst? Er presste sich ganz eng an die Wand hinter ihm und hoffte, der Verfolger würde ihn übersehen, als plötzlich neben ihm mit kaum wahrnehmbarem Quietschen eine Tür im nächtlichen Wind hin und her schwang. Könnte dieses der flehentlich herbeigesehnte Ausweg aus seinem Dilemma sein? Langsam bewegte er sich auf die offene Tür zu, immer dicht an die Mauer gepresst. Würde diese Tür seine Rettung werden?

A Anhang

A.1 Quelltexte

cpu.c aus Linux 2.6.16

```

1  /* CPU control.
2   * (C) 2001, 2002, 2003, 2004 Rusty Russell
3   *
4   * This code is licenced under the GPL.
5   */
6  #include <linux/proc_fs.h>
7  #include <linux/smp.h>
8  #include <linux/init.h>
9  #include <linux/notifier.h>
10 #include <linux/sched.h>
11 #include <linux/unistd.h>
12 #include <linux/cpu.h>
13 #include <linux/module.h>
14 #include <linux/kthread.h>
15 #include <linux/stop_machine.h>
16 #include <asm/semaphore.h>
17
18 /* This protects CPUs going up and down... */
19 static DECLARE_MUTEX(cpucontrol);
20
21 static struct notifier_block *cpu_chain;
22
23 #ifdef CONFIG_HOTPLUG_CPU
24 static struct task_struct *lock_cpu_hotplug_owner;
25 static int lock_cpu_hotplug_depth;
26
27 static int __lock_cpu_hotplug(int interruptible)
28 {

```

```

29     int ret = 0;
30
31     if (lock_cpu_hotplug_owner != current) {
32         if (interruptible)
33             ret = down_interruptible(&cpucontrol);
34         else
35             down(&cpucontrol);
36     }
37
38     /*
39      * Set only if we succeed in locking
40      */
41     if (!ret) {
42         lock_cpu_hotplug_depth++;
43         lock_cpu_hotplug_owner = current;
44     }
45
46     return ret;
47 }
48
49 void lock_cpu_hotplug(void)
50 {
51     __lock_cpu_hotplug(0);
52 }
53 EXPORT_SYMBOL_GPL(lock_cpu_hotplug);
54
55 void unlock_cpu_hotplug(void)
56 {
57     if (--lock_cpu_hotplug_depth == 0) {
58         lock_cpu_hotplug_owner = NULL;
59         up(&cpucontrol);
60     }
61 }
62 EXPORT_SYMBOL_GPL(unlock_cpu_hotplug);
63
64 int lock_cpu_hotplug_interruptible(void)
65 {
66     return __lock_cpu_hotplug(1);
67 }
68 EXPORT_SYMBOL_GPL(lock_cpu_hotplug_interruptible);
69 #endif /* CONFIG_HOTPLUG_CPU */
70
71 /* Need to know about CPUs going up/down? */
72 int register_cpu_notifier(struct notifier_block *nb)
73 {
74     int ret;
75
76     if ((ret = lock_cpu_hotplug_interruptible()) != 0)
77         return ret;
78     ret = notifier_chain_register(&cpu_chain, nb);
79     unlock_cpu_hotplug();
80     return ret;
81 }

```

```

82 EXPORT_SYMBOL(register_cpu_notifier);
83
84 void unregister_cpu_notifier(struct notifier_block *nb)
85 {
86     lock_cpu_hotplug();
87     notifier_chain_unregister(&cpu_chain, nb);
88     unlock_cpu_hotplug();
89 }
90 EXPORT_SYMBOL(unregister_cpu_notifier);
91
92 #ifdef CONFIG_HOTPLUG_CPU
93 static inline void check_for_tasks(int cpu)
94 {
95     struct task_struct *p;
96
97     write_lock_irq(&tasklist_lock);
98     for_each_process(p) {
99         if (task_cpu(p) == cpu &&
100             (!cputime_eq(p->utime, cputime_zero) ||
101              !cputime_eq(p->stime, cputime_zero)))
102             printk(KERN_WARNING "Task_%s_(pid=%d)_is_on_cpu_%d\
103             uuuuuuuu(state=%ld, flags=%lx)\n",
104                p->comm, p->pid, cpu, p->state, p->flags);
105     }
106     write_unlock_irq(&tasklist_lock);
107 }
108
109 /* Take this CPU down. */
110 static int take_cpu_down(void *unused)
111 {
112     int err;
113
114     /* Ensure this CPU doesn't handle any more interrupts. */
115     err = __cpu_disable();
116     if (err < 0)
117         return err;
118
119     /* Force idle task to run as soon as we yield: it should
120        immediately notice cpu is offline and die quickly. */
121     sched_idle_next();
122     return 0;
123 }
124
125 int cpu_down(unsigned int cpu)
126 {
127     int err;
128     struct task_struct *p;
129     cpumask_t old_allowed, tmp;
130
131     if ((err = lock_cpu_hotplug_interruptible()) != 0)
132         return err;
133
134     if (num_online_cpus() == 1) {

```

```

135         err = -EBUSY;
136         goto out;
137     }
138
139     if (!cpu_online(cpu)) {
140         err = -EINVAL;
141         goto out;
142     }
143
144     err = notifier_call_chain(&cpu_chain, CPU_DOWN_PREPARE,
145                             (void *) (long) cpu);
146     if (err == NOTIFY_BAD) {
147         printk("%s: attempt to take down CPU_%u failed\n",
148             __FUNCTION__, cpu);
149         err = -EINVAL;
150         goto out;
151     }
152
153     /* Ensure that we are not runnable on dying cpu */
154     old_allowed = current->cpus_allowed;
155     tmp = CPU_MASK_ALL;
156     cpu_clear(cpu, tmp);
157     set_cpus_allowed(current, tmp);
158
159     p = __stop_machine_run(take_cpu_down, NULL, cpu);
160     if (IS_ERR(p)) {
161         /* CPU didn't die: tell everyone. Can't complain. */
162         if (notifier_call_chain(&cpu_chain, CPU_DOWN_FAILED,
163                                 (void *) (long) cpu) == NOTIFY_BAD)
164             BUG();
165
166         err = PTR_ERR(p);
167         goto out_allowed;
168     }
169
170     if (cpu_online(cpu))
171         goto out_thread;
172
173     /* Wait for it to sleep (leaving idle task). */
174     while (!idle_cpu(cpu))
175         yield();
176
177     /* This actually kills the CPU. */
178     __cpu_die(cpu);
179
180     /* Move it here so it can run. */
181     kthread_bind(p, get_cpu());
182     put_cpu();
183
184     /* CPU is completely dead: tell everyone. Too late to complain. */
185     if (notifier_call_chain(&cpu_chain, CPU_DEAD, (void *) (long) cpu)
186         == NOTIFY_BAD)
187         BUG();

```

```

188     check_for_tasks(cpu);
189
190 out_thread:
191     err = kthread_stop(p);
192 out_allowed:
193     set_cpus_allowed(current, old_allowed);
194 out:
195     unlock_cpu_hotplug();
196     return err;
197 }
198 #endif /*CONFIG_HOTPLUG_CPU*/
199
200 int __devinit cpu_up(unsigned int cpu)
201 {
202     int ret;
203     void *hcpu = (void *) (long)cpu;
204
205     if ((ret = lock_cpu_hotplug_interruptible()) != 0)
206         return ret;
207
208     if (cpu_online(cpu) || !cpu_present(cpu)) {
209         ret = -EINVAL;
210         goto out;
211     }
212 }
213

```

```

214     ret = notifier_call_chain(&cpu_chain, CPU_UP_PREPARE, hcpu);
215     if (ret == NOTIFY_BAD) {
216         printk("%s: attempt to bring up CPU %u failed\n",
217             __FUNCTION__, cpu);
218         ret = -EINVAL;
219         goto out_notify;
220     }
221
222     /* Arch-specific enabling code. */
223     ret = __cpu_up(cpu);
224     if (ret != 0)
225         goto out_notify;
226     if (!cpu_online(cpu))
227         BUG();
228
229     /* Now call notifier in preparation. */
230     notifier_call_chain(&cpu_chain, CPU_ONLINE, hcpu);
231
232 out_notify:
233     if (ret != 0)
234         notifier_call_chain(&cpu_chain, CPU_UP_CANCELED, hcpu);
235 out:
236     unlock_cpu_hotplug();
237     return ret;
238 }

```


Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum

Unterschrift