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 IV

Tabellenverzeichnis V

Symbolverzeichnis VI

1 Einleitung 1

2 Diskretisierung der Poisson-Gleichung im $\mathbb{R}^2$ 3
2.1 Definition (Poisson-Gleichung) 3
2.2 Bemerkungen 4
2.3 Finite Differenzen-Methode für die Poisson-Gleichung 4
2.3.1 (Zentraler) Differenzenquotient zweiter Ordnung 4
2.3.2 Diskretisierung von Ω 5

3 Iterative Lösungsvefahren für lineare Gleichungssysteme 9
3.1 Das Jacobi-Verfahren (Gesamtschrittverfahren) 9
3.1.1 Das allgemeine Jacobi-Iterationsverfahren 9
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
3.2.2 Das Gauss-Seidel-Iterationsverfahren für die Poisson-Gleichung . . 12
3.3 Warum Einzelschritt- und Gesamtschrittverfahren? 12
3.4 Das Verfahren der konjugierten Gradienten 13
3.4.1 Definition (A-orthogonal) 13
3.4.2 Satz - Minimierungsfunktion 13
3.4.3 Lemma - (A-orthogonaler) Projektionssatz 14

3.4.4 Allgemeiner Algorithmus der konjugierten Gradienten 15
3.4.5 Numerischer Algorithmus der konjugierten Gradienten 16
3.4.6 Satz - Verallgemeinerung des Startvektors 17
3.4.7 Lemma (Zusammenhang zu Krylovräumen) 18

4 Mehrgitterverfahren 19
4.1 Grundideen 19
4.1.1 Beweis der Residuumsgleichung 20
4.2 Das Zweigitter-Verfahren 20

5 Zusammenfassung 21

A Anhang 23
A.1 Quelltexte 23

Abbildungsverzeichnis

2.1 5-Punkt-Differenzenstern im Gitter 6
2.2 Gitter 6

Tabellenverzeichnis

5.1 eine sinnlose Tabelle 21
5.2 eine kompliziertere Tabelle 21

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. Aus diesem Grund ist das Interesse an ihnen sehr groß. Solche Gleichungen zu lösen ist allerdings nicht immer möglich, oder sehr aufwendig. Eine der bekanntesten PDGs ist die Poisson-Gleichung – eine elliptische partielle Differentialgleichung zweiter Ordnung. Sie wurde vom Mathematiker und Physik Simeon Denis Poisson aufgestellt und findet vor Allem in der Physik Anwendung, da sie dem elektrostatischem Potential und dem Gravitationspotential genügt. Methoden aus der numerischen Mathematik ermöglichen uns das Lösen von partiellen Differentialgleichungen mittels computerbasierten Algorithmen. Hierbei wird jedoch nicht das Ergebnis direkt ausgerechnet, sondern versucht eine exakte Lösung zu approximieren. Diese Approximationen werden mittels Computerprogrammen realisiert und aus diesem Grund ist ein effizientes, numerisch stabiles Verfahren unabdingbar. Im folgenden werden wir zwei Methoden kennen lernen, die genau diese Kriterien erfüllen. Um nun die Lösung einer partiellen Differentialgleichung 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 führen wir die PDG, die auf einem gewissen Gebiet definiert ist auf das Einheitsquadrat zurück, legen ein Gitter darüber und erhalten durch diese Diskretisierung ein lineares Gleichungssystem. Da für lineare Gleichungssysteme direkte Lösungsverfahren, wie beispielsweise der Gauß-Algorithmus, existieren, bekommen wir unter Maschinengenauigkeit ein exaktes Ergebnis für die PDG. Wir werden allerdings sehen, dass der Rechenaufwand für große Systeme ungünstig ist. Eine weitere Möglichkeit zur Lösung sind iterative Verfahren. Diese haben nicht nur den Vorteil, dass sie nun mit hohen Dimension (z.B. Einer yxy-Matrix) kein Problem mehr haben, sondern auch wesentlich schneller gegen die exakte Approximation konvergieren. Da es eine Vielzahl an iterativen Lösungsverfahren gibt, werden wir uns hier auf das Verfahren der konjugierten Gradienten (mit Vorkonditionierung) und Mehrgittermethoden beschränken. Beide Verfahren haben gewisse Vorzüge, die wir gegeneinander abwiegen und so einen Vergleich der Verfahren erhalten werden. Abschließend werden wir uns noch der Implementierung beider Verfahren widmen und ein konkretes Beispiel sehen.

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

Um nun die Poisson-Gleichung zu diskretisieren, werden wir diese zunächst definieren. Außerdem werden wir die Methode der finitien Differenzen einführen, um dann ein Gleichungssystem der Form $\mathbf{A}x = b$ 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

$$\begin{aligned} -\Delta u(x,y) &= f(x,y) \text{ in } \Omega && (2.1) \\ u(x,y) &= g(x,y) \text{ in } \partial\Omega && (2.2) \end{aligned}$$

löst. Dabei bezeichnet $\Delta := \sum\limits_{k=1}^{n} \frac{\partial^2}{\partial x_k^2}$ den Laplace-Operator. Für die Poisson-Gleichung im $\mathbb{R}^2$ gilt dann:

$$\begin{aligned} -\Delta u(x,y) &= \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y) \text{ in } \Omega && (2.3) \\ u(x,y) &= g(x,y) \text{ in } \partial\Omega && (2.4) \end{aligned}$$

(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

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

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

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:

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

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

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

Diese Näherungen nennt man auch (zentralen) Differenzenquotienten der zweiten Ableitung, wobei $O(h^2)$ ein Term zweiter Ordnung ist und später vernachlässigt wird/werden kann. Somit erhalten wir für $\Delta u(x,y)$ die Näherung

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

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

$$u(x_i, y_j) := u(ih, jh) \tag{2.10}$$

und für Ω lässt sich ein Ω_h einführen, so dass:

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

Betrachten wir nun noch den Rand von Ω bzw. Ω_h, dann gilt:

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

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

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

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

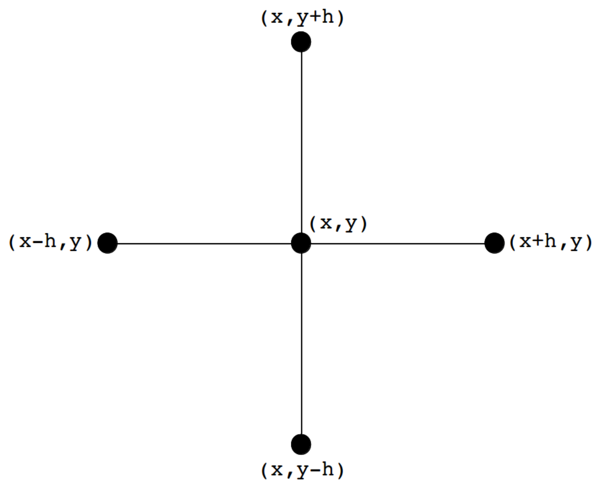

Abbildung 2.1: 5-Punkt-Differenzenstern im Gitter

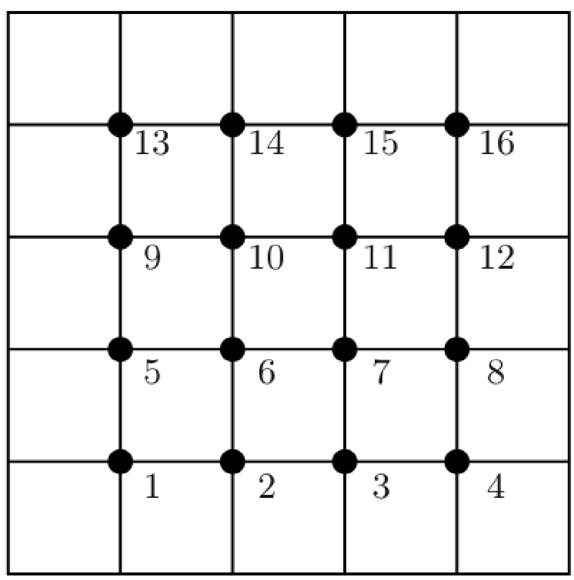

Abbildung 2.2: (Lexikographische) Nummerierung von $\Omega = (0,1)^2$ mit $n = 5$

Nummeriert man nun alle Gitterpunkte des Gitters fortlaufend von links unten nach rechts oben durch (Abbildung 2.3.2) 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:

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

Wobei $\mathbf{Id} \in \mathbb{R}^{(n-1)\times(n-1)}$ die Identität - also die Einheitsmatrix - ist und für alle $i = 0, .., n$ gilt:

$$A_i = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \tag{2.15}$$

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

Um nun auf ein lineares Gleichungssystem der Form $\mathbf{A}u = b$ zu kommen, muss natürlich noch die rechte Seite, also das b 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 somit nicht genähert werden müssen. Jedoch muss zu jeder Komponente in b, die einen Randpunkt als Nachbarn hat, dieser aufaddiert werden. Dies führt uns auf folgende rechte Seite:

$$b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{pmatrix} \tag{2.16}$$

wobei gilt

$$b_1 = \begin{pmatrix} f(h,h) + h^{-2}(g(h,0) + g(0,h)) \\ f(2h,h) + h^{-2}(g(2h,0)) \\ \vdots \\ f(1-2h,h) + h^{-2}(g(1-2h,0)) \\ f(1-h,h) + h^{-2}(g(1-h,0) + g(0,1-h)) \end{pmatrix} \tag{2.17}$$

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

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

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

3 Iterative Lösungsvefahren 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 und unstabil. Ein wesentlicher Bestandteil der numerischen Mathematik sind iterative Verfahren zu 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) und dem CG-Verfahren (samt 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 $n = (N-1) \cdot (N-1)$.

3.1.1 Das allgemeine Jacobi-Iterationsverfahren

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

$$A = D - L - U \tag{3.1}$$

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 $\mathbf{A}u = b$:

$$Au = b \Leftrightarrow (D - L - U)u = b \Leftrightarrow Du = (L + U)u + b \tag{3.2}$$

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

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

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 $k = 1, 2, ...$ berechne für $i = 1, ..., n$:

$$u_i^k = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} u_i^{k-1}) \tag{3.4}$$

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:

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

Löst man diese Gleichung nun nach $u(x_i, y_j)$ auf und geht jeden Gitterpunkt Schritt-für-Schritt durch, so erhält man folgende Iterationsvorschrift für das Jacobi-Verfahren:
Für $k = 1, 2, ...$ berechne mit Startvektor $u^0 \in \mathbb{R}^n$
Für $i, j = 1, ..., N - 1$

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

Wie nun zu sehen ist, beträgt der Rechenaufwand pro Iterationsschritt lediglich $\mathcal{O}((N-1)\cdot(N-1)) = \mathcal{O}(n)$ Schritte für die Poisson-Gleichung. Zudem ist zu erwähnen, dass einige der $u_{i,j}$ Informationen über den Rand enthalten, die ja aus der Voraussetzung bekannt sind.

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 **A** wie folgt zerlegen:

$$A = D - L - U \tag{3.7}$$

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

$$Au = b \Leftrightarrow (D - L - U)u = b \Leftrightarrow (D - L)u = Uu + b \tag{3.8}$$

Daraus können wir nun folgende Iterationsvorschrift ableiten:

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

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

$$\sum_{j=1}^{i} a_{ij}u_j^{k+1} = -\sum_{j=i+1}^{n} a_{ij}u_j^k + b_i \tag{3.10}$$

Formt man Gleichung 3.10 um, so erhält man den Algorithmus des Gauss-Seidel-Verfahrens mit Startvektor $u^0 \in \mathbb{R}$:
Für $k = 1, 2, ...$ berechne für $i = 1, ..., n$

$$u_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}u_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}u_j^k) \tag{3.11}$$

Wie zu sehen ist, verwendet dieser Algorithmus dieses mal nicht nur die Werte aus dem vorigen Iterationsschritt, sondern auch welche aus dem der gerade berechnet wird. Jedoch sind dies bereits neu berechnete Werte und werde in diesem Iterationsschritt nicht mehr geändert.Das Gauss-Seidel-Verfahren wartet ebenfalls mit einem Rechaufwand 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 Abbildung 2.3.2 ansehen. Hierbei ist ersichtlich, dass $u_{i-1,j}$ und $u_{i,j-1}$ bereits neu berechnet wurde. $u_{i+1,j}$ und $u_{i,j+1}$ stammen noch aus dem vorigen Iterationsschritt.

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

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

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 zu Beginn der Iteration aus bzw. reduzieren diese stark. Darum finden sie besonders große Verwendung in den Mehrgittermethoden, die wir später kennen lernen werden.

3.4 Das Verfahren der konjugierten Gradienten

Das Verfahren der konjugierten Gradienten wurde 1952 von Heestens und Stiefel erstmals vorgestellt und erfreut sich großer Beliebtheit. Diese rührt daher, da das Verfahren eine sehr schnelle Konvergenz hat und numerisch äußerst stabil ist.
Das Verfahren der konjugierten Gradienten (oder CG-Verfahren) ist eine Krylov-Unterraum-Methode und gehört zu den Projektionsverfahren. Krylovräume sind, wie wir sehen werden, (Unter-)Räume die durch eine Matrizen und Residuenvektoren aufgespannt werden. Innerhalb dieser Räume wird ein orthogonaler Vektor gesucht, auf den Krylovraum projeziert und anschließend orthonormalisiert. Den Unterschied - nicht nur in der Konvergenzgeschwindigkeit - zum Gradientenabstiegsverfahren machen beim CG-Verfahren die konjugierten Richtungen. Sie kommen durch spezielle - A-orthogonale - Projektionen zu stande. Um dies nun mathematisch sauber zu formulieren, definieren wir zunächst, was A-orthogonal oder auch konjugiert heißt:

3.4.1 Definition (A-orthogonal)

Sei $\mathbf{A}$ 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.

Bemerkung:

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

Wozu die A-Orthogonalität dient, wird uns der nächste Satz zeigen:

3.4.2 Satz - Minimierungsfunktion

Sei $\mathbf{A} \in \mathbb{R}^{n \times n}$ s.p.d. und

$$f(x) := \frac{1}{2} x^T \mathbf{A} x - b^T x, \tag{3.13}$$

wobei $b, x \in \mathbb{R}^n$. Dann gilt:

f hat ein eindeutig bestimmtes Minimum und

$$Ax^* = b \Longleftrightarrow f(x^*) = \min_{x \in \mathbb{R}^n} f(x) \tag{3.14}$$

Beweis siehe z.B. Dahmen/Reusken...

Bei der Minimierung der Funktion Gleichung 3.13 werden beim Gradientenverfahren die Richtungen des steilsten Abstiegs gesucht. Im CG-Verfahren geschieht im wesentlichen das Gleiche, allerdings bewirkt die Matrix **A**, dass wir unser Ziel - das Miniumum der Funktion - schneller erreichen.
Illustriert wird dies durch Bild (bla). Hier ist gut zu sehen, dass beim Gradientenverfahren die Höhenlinien auf elliptischen Bahnen liegen. Durch die Matrix A, also die A-Orthogonalität, werden die Ellipsen gestaucht zu Kreisen.

Folgendes Lemma wird illustrieren, wie wir uns diese neu gewonne Erkenntnis nun zu Nutze machen werden.

3.4.3 Lemma - (A-orthogonaler) Projektionssatz

Sei U_k ein k-dimensionaler Teilraum des $\mathbb{R}^n$ $(k \leq n)$, und $p^0, p^1, ..., p^{k-1}$ 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 $u^k \in U_k$:

$$\|u^k - v\|_A = \min_{u \in U_k} \|u - v\|_A \tag{3.15}$$

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

$$P_{U_{k,\langle\cdot,\cdot\rangle}}(v) = \mathbf{u}^k = \sum_{j=0}^{k-1} \frac{\langle v, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \tag{3.16}$$

Der Beweis zu diesem Lemma folgt direkt aus dem Projektionssatz. Man sucht einen Vektor v in U_{k+1}. Man wählt nun ein beliebiges $v \in U_{k+1}$ und minimiert über alle $u^k \in U_k$.

Die optimale Lösung ist dann der gesucht Vektor v. Hier wird in die Minimierungsfunkton jedoch noch die Matrix **A** hinzugenommen.
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.

3.4.4 Allgemeiner Algorithmus der konjugierten Gradienten

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

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

$$U_1 := span\{r^0\}, \text{wobei } r^0 = b - Ax^0 \tag{3.17}$$

1. Bestimme eine A-orthogonale Basis

$$p^0, ..., p^{k-1} \text{ von } U_k \tag{3.18}$$

2. Bestimme eine Näherungslösung x^k, so dass

$$\|x^k - x^*\|_A = \min_{u \in U_k} \|x - x^*\|_A \tag{3.19}$$

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

$$x^k = \sum_{j=0}^{k-1} \frac{\langle x^*, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \tag{3.20}$$

3. Erweitere den Teilraum U_k und berechne das iterierte Residuum

$$U_{k+1} := span\{p^0, ..., p^{k-1}, r^k\} \text{ wobei } r^k := b - Ax^k \tag{3.21}$$

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:

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 initialisisert und ein Residuum bestimmen muss. Hierfür ist ein Startvektor x^0 beliebig zu wählen (**??**). Das Residuum wird durch $r^0 := b - Ax^0$ definiert.

zu 1. Um eine A-orthogonale Basis von den U_k zu bestimmen ist im wesentlichen eine (A-) Orthogonalisierungverfahren notwendig, welches hier auch angewandt wird.

zu 2. Hier wird das in Unterabschnitt 3.4.3 bereits besprochene Verfahren angewandt, um eine neue Näherungslösung in U_k zu bestimmen.

zu 3. Hier soll im wesentlich das Gleiche geschehen in wie in Schritt 0. Lediglich wird nun das $k - te$ Resdiduum 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.

3.4.5 Numerischer Algorithmus der konjugierten Gradienten

Gegeben ist eine symmetrisch positiv definitie Matrix $\mathbf{A} \in \mathbb{R}^n$. 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 $k = 1, 2, ...,$ falls $r^{k-1} \neq 0$ berechne:

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

Das der Startvektor beliebig ist (gilt bei allen iterativen Verfahren) wollen wir im Folgenden noch zeigen und beweisen.

3.4.6 Satz - Verallgemeinerung des Startvektors

Das Verfahren der konjugierten Gradienten ist unabhängig von der Wahl des Startvektors x^0.

Beweis:

Zu lösen: $\mathbf{A}x^* = b$.
Sei $x^0 \neq 0$. Definiere für das transformierte System $\mathbf{A}\tilde{x} = \tilde{b}$, $\tilde{x} := x^* - x^0$ und $\tilde{b} := b - Ax^0$
$\Longrightarrow A\tilde{x} = A(x^* - x^0) = b - Ax^0 = r^0$
Sei nun: $\tilde{x}^0 = 0$ Startvektor mit Residuum $\tilde{r}$.
$\Longrightarrow \tilde{x}^k = x^k - x^0 \Longrightarrow x^k = \tilde{x}^k + x^0$
$\Longrightarrow \tilde{r}^k = \tilde{b} - A\tilde{x}^k = b - Ax^0 - A\tilde{x}^k$
$= b - A(x^0 - \tilde{x}^k) = b - Ax^k = r^k$
$\Longrightarrow \tilde{r}^k = r^k$

3.4.7 Lemma (Zusammenhang zu Krylovräumen)

Man kann U_k auch in folgender Form schreiben:

$$U_k := span\{r^0, r^1, ..., r^{k-1}\} = span\{p^0, p^1, ..., p^{k-1}\} = span\{r^0, Ar^0, ..., A^{k-1}r^0\} \tag{3.22}$$

Beweis:

Per Induktion über k (klar für k=0):
Induktionsvoraussetzung:

$$U_k := span\{r^0, r^1, ..., r^{k-1}\} = span\{p^0, p^1, ..., p^{k-1}\} = span\{r^0, Ar^0, ..., A^{k-1}r^0\}$$
$$\implies k \longrightarrow k+1 : U_{k+1}$$
$$r^k = r^{k-1} - \alpha_{k-1}Ap^{k-1}$$
$$p^{k-1} \in U_k = span\{r^0, ..., A^{k-1}r^0\}$$
$$da \;\; p^{k-1} = (\sum_{i=0}^{k-1} \sigma_i A^i)r^0$$
$$\implies Ap^{k-1} = (\sum_{i=0}^{k-1} \sigma_i A^{i+1})r^0$$
$$= \sigma_0 Ar^0 + ... + \sigma_{k-1}A^k r^0$$
$$\implies r_k \in U_{k+1}$$

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. Residuumsgleichung
Die für diesen Algorithmus wichtige Residuumsgleichung lautete:

$$\mathbf{A}\epsilon^k = -r^k \tag{4.1}$$

Wobei die Lösung von $\mathbf{A}\epsilon^k = -r^k$, wobei $\epsilon^k = 0$ äquivalent zur Lösung von $\mathbf{A}u = b$ ist.

4.1.1 Beweis der Residuumsgleichung

Das Residuum ist an der $k-ten$ Stelle definiert als

$$r^k = b - \mathbf{A}u^k \tag{4.2}$$

Der Fehler

$$\epsilon = u^* - u^k \tag{4.3}$$

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

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

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

Gauss-Seidel- und Jacobi-Verfahren löschen also hochfrequente Fehler in den ersten Itertionsschritten 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 inAbschnitt 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 $\epsilon^k = u^* - u^k$ verschwunden. Nun berechnet man im $k-ten$ Schritt das Residuum r^k und führt für das äquivalente lineare Gleichungssystem $\mathbf{A}\epsilon^k = r^k$, wobei $\epsilon^k = 0$ gilt, l Iterationsschritte aus. So erhalten wir eine Näherung des Fehlers ϵ^k.
Stellt man Gleichung 4.3 um, berechnet also $\epsilon^k + u^k$, 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 Das Zweigitter-Verfahren

5 Zusammenfassung

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 auftauschen soll

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

```
/* CPU control.
 * (C) 2001, 2002, 2003, 2004 Rusty Russell
 *
 * This code is licenced under the GPL.
 */
#include <linux/proc_fs.h>
#include <linux/smp.h>
#include <linux/init.h>
#include <linux/notifier.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/cpu.h>
#include <linux/module.h>
#include <linux/kthread.h>
#include <linux/stop_machine.h>
#include <asm/semaphore.h>

/* This protects CPUs going up and down... */
static DECLARE_MUTEX(cpucontrol);

static struct notifier_block *cpu_chain;

#ifdef CONFIG_HOTPLUG_CPU
static struct task_struct *lock_cpu_hotplug_owner;
static int lock_cpu_hotplug_depth;

static int __lock_cpu_hotplug(int interruptible)
{
  int ret = 0;

  if (lock_cpu_hotplug_owner != current) {
    if (interruptible)
      ret = down_interruptible(&cpucontrol);
    else
      down(&cpucontrol);
  }

  /*
   * Set only if we succeed in locking
   */
  if (!ret) {
    lock_cpu_hotplug_depth++;
    lock_cpu_hotplug_owner = current;
  }

  return ret;
}

void lock_cpu_hotplug(void)
{
  __lock_cpu_hotplug(0);
}
EXPORT_SYMBOL_GPL(lock_cpu_hotplug);

void unlock_cpu_hotplug(void)
{
  if (--lock_cpu_hotplug_depth == 0) {
    lock_cpu_hotplug_owner = NULL;
    up(&cpucontrol);
  }
}
EXPORT_SYMBOL_GPL(unlock_cpu_hotplug);

int lock_cpu_hotplug_interruptible(void)
{
  return __lock_cpu_hotplug(1);
}
EXPORT_SYMBOL_GPL(lock_cpu_hotplug_interruptible);
#endif  /* CONFIG_HOTPLUG_CPU */

/* Need to know about CPUs going up/down? */
int register_cpu_notifier(struct notifier_block *nb)
{
  int ret;

  if ((ret = lock_cpu_hotplug_interruptible()) != 0)
    return ret;
  ret = notifier_chain_register(&cpu_chain, nb);
  unlock_cpu_hotplug();
  return ret;
}
```

```c
EXPORT_SYMBOL(register_cpu_notifier);

void unregister_cpu_notifier(struct notifier_block *nb)
{
  lock_cpu_hotplug();
  notifier_chain_unregister(&cpu_chain, nb);
  unlock_cpu_hotplug();
}
EXPORT_SYMBOL(unregister_cpu_notifier);

#ifdef CONFIG_HOTPLUG_CPU
static inline void check_for_tasks(int cpu)
{
  struct task_struct *p;

  write_lock_irq(&tasklist_lock);
  for_each_process(p) {
    if (task_cpu(p) == cpu &&
        (!cputime_eq(p->utime, cputime_zero) ||
         !cputime_eq(p->stime, cputime_zero)))
      printk(KERN_WARNING "Task %s (pid = %d) is on cpu %d\
        (state = %ld, flags = %lx) \n",
         p->comm, p->pid, cpu, p->state, p->flags);
  }
  write_unlock_irq(&tasklist_lock);
}

/* Take this CPU down. */
static int take_cpu_down(void *unused)
{
  int err;

  /* Ensure this CPU doesn't handle any more interrupts. */
  err = __cpu_disable();
  if (err < 0)
    return err;

  /* Force idle task to run as soon as we yield: it should
     immediately notice cpu is offline and die quickly. */
  sched_idle_next();
  return 0;
}

int cpu_down(unsigned int cpu)
{
  int err;
  struct task_struct *p;
  cpumask_t old_allowed, tmp;

  if ((err = lock_cpu_hotplug_interruptible()) != 0)
    return err;

  if (num_online_cpus() == 1) {
    err = -EBUSY;
    goto out;
  }

  if (!cpu_online(cpu)) {
    err = -EINVAL;
    goto out;
  }

  err = notifier_call_chain(&cpu_chain, CPU_DOWN_PREPARE,
          (void *)(long)cpu);
  if (err == NOTIFY_BAD) {
    printk("%s: attempt to take down CPU %u failed\n",
        __FUNCTION__, cpu);
    err = -EINVAL;
    goto out;
  }

  /* Ensure that we are not runnable on dying cpu */
  old_allowed = current->cpus_allowed;
  tmp = CPU_MASK_ALL;
  cpu_clear(cpu, tmp);
  set_cpus_allowed(current, tmp);

  p = __stop_machine_run(take_cpu_down, NULL, cpu);
  if (IS_ERR(p)) {
    /* CPU didn't die: tell everyone.  Can't complain. */
    if (notifier_call_chain(&cpu_chain, CPU_DOWN_FAILED,
        (void *)(long)cpu) == NOTIFY_BAD)
      BUG();

    err = PTR_ERR(p);
    goto out_allowed;
  }

  if (cpu_online(cpu))
    goto out_thread;

  /* Wait for it to sleep (leaving idle task). */
  while (!idle_cpu(cpu))
    yield();

  /* This actually kills the CPU. */
  __cpu_die(cpu);

  /* Move it here so it can run. */
  kthread_bind(p, get_cpu());
  put_cpu();

  /* CPU is completely dead: tell everyone.  Too late to complain. */
  if (notifier_call_chain(&cpu_chain, CPU_DEAD, (void *)(long)cpu)
      == NOTIFY_BAD)
    BUG();
```

```

	check_for_tasks(cpu);

out_thread:
	err = kthread_stop(p);
out_allowed:
	set_cpus_allowed(current, old_allowed);
out:
	unlock_cpu_hotplug();
	return err;
}
#endif /*CONFIG_HOTPLUG_CPU*/

int __devinit cpu_up(unsigned int cpu)
{
	int ret;
	void *hcpu = (void *)(long)cpu;

	if ((ret = lock_cpu_hotplug_interruptible()) != 0)
		return ret;

	if (cpu_online(cpu) || !cpu_present(cpu)) {
		ret = -EINVAL;
		goto out;
	}

	ret = notifier_call_chain(&cpu_chain, CPU_UP_PREPARE, hcpu);
	if (ret == NOTIFY_BAD) {
		printk("%s:␣attempt␣to␣bring␣up␣CPU␣%u␣failed\n",
				__FUNCTION__, cpu);
		ret = -EINVAL;
		goto out_notify;
	}

	/* Arch-specific enabling code. */
	ret = __cpu_up(cpu);
	if (ret != 0)
		goto out_notify;
	if (!cpu_online(cpu))
		BUG();

	/* Now call notifier in preparation. */
	notifier_call_chain(&cpu_chain, CPU_ONLINE, hcpu);

out_notify:
	if (ret != 0)
		notifier_call_chain(&cpu_chain, CPU_UP_CANCELED, hcpu);
out:
	unlock_cpu_hotplug();
	return ret;
}
```

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