mm

Universität Regensburg

Fachbereich Mathematik

Abbildung 0.1:

Bachelorarbeit

im Studiengang Computational Science

Iterative Verfahren zur Lösung der diskretisierten Poisson Gleichung von

Michael Bauer

Betreuer: Prof. Dr. Harald Garcke

Eingereicht am: Datum

Inhaltsverzeichnis

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 $\mathbb{R}^2$

2.1 Definition (Poisson-Gleichung)

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

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

löst. Dabei seien $f : \Omega \to \mathbb{R}$ und $g : \partial\Omega \to \mathbb{R}$ 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:

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

Gleichung 2.2 bzw. Gleichung 2.4 nennt man Dirichlet-Randbedingung.

Beachte: $\partial_{xx} u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2}$.

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

2.2 Finite Differenzen-Methode und Diskretisierung von Ω

Zunächst wollen wir nun den zentralen Differenzenquotienten (zweiter Ordnung) einführen. Diesen verwenden wir dann bei der Diskretisierung

unseres Gitters Ω. Die Matrix, die wir für den Fall der Poisson Gleichung erhalten werden, soll dann auch noch auf ihre Eigenschaften untersucht werden.

2.2.1 Zentraler Differenzenquotient zweiter Ordnung

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

$$u(x+h,y) = \sum_{k=0}^{n} \frac{h^k}{k!} \frac{\partial^k u(x,y)}{\partial x^k} \approx u(x,y) + h\partial_x u(x,y) + \frac{h^2}{2!}\partial_{xx} u(x,y) + \mathcal{O}(h^3), \tag{2.5}$$

$$u(x-h,y) = \sum_{k=0}^{n} (-1)^k \frac{h^k}{k!} \frac{\partial^k u(x,y)}{\partial y^k} \approx u(x,y) - h\partial_x u(x,y) + \frac{h^2}{2!}\partial_{xx} u(x,y) - \mathcal{O}(h^3). \tag{2.6}$$

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

$$u(x,y+h) = \sum_{k=0}^{n} \frac{h^k}{k!} \frac{\partial^k u(x,y)}{\partial y^k} \approx u(x,y) + h\partial_y u(x,y) + \frac{h^2}{2!}\partial_{yy} u(x,y) + \mathcal{O}(h^3), \tag{2.7}$$

$$u(x,y-h) = \sum_{k=0}^{n} (-1)^k \frac{h^k}{k!} \frac{\partial^k u(x,y)}{\partial y^k} \approx u(x,y) - h\partial_y u(x,y) + \frac{h^2}{2!}\partial_{yy} u(x,y) - \mathcal{O}(h^3). \tag{2.8}$$

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

$$\partial_{xx} u(x,y) + O(h^2) = \frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2}. \tag{2.9}$$

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

$$\partial_{yy} u(x,y) + O(h^2) = \frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2}. \tag{2.10}$$

Diese Näherungen nennt man den **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

$$-\Delta u(x,y) \approx \frac{u(x-h,y) - u(x+h,y) + 4u(x,y) - u(x,y-h) - u(x,y+h)}{h^2}. \tag{2.11}$$

2.2.2 Diskretisierung von Ω

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

Für $i, j = 1, ..., N$ kann man dann $u(x,y)$ auch schreiben als:

$$u(x_i, y_j) = u(ih, jh). \tag{2.12}$$

Ω fassen wir als Ω_h auf, so dass:

$$\Omega_h := \{u(ih, jh) | 1 \leq i, j \leq N\}. \tag{2.13}$$

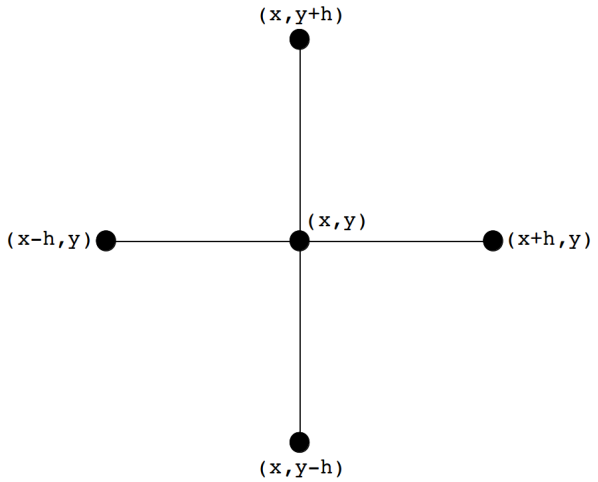

Abbildung 2.1: 5-Punkt-Differenzenstern im Gitter

Mit Gleichung 2.11 fassen wir $\Delta u(x,y) = \Delta_h u(x,y)$ in diskretisierter Form auf:

$$-\Delta_h u(x,y) = \frac{u(x-h,y) - u(x+h,y) + 4u(x,y) - u(x,y-h) - u(x,y+h)}{h^2}. \tag{2.14}$$

für alle $(x,y) \in \Omega_h$.
Man stellt $\Delta_h u(x,y)$ auch als 5-Punkt-Differenzenstern (Abbildung 2.2.2) in der Form

$$[-\Delta_h]_\xi = \frac{1}{h^2} \begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix}, \xi \in \Omega_h \tag{2.15}$$

dar. (Dahmen Reusken)

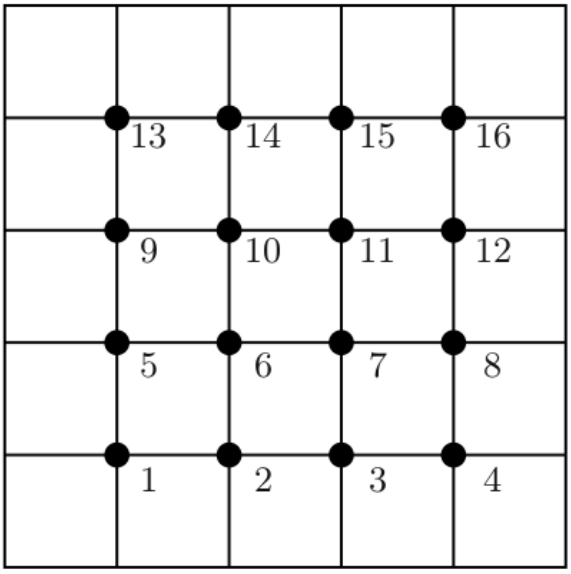

Abbildung 2.2: Nummerierung der Punkte von $\Omega = (0,1)^2$ mit $m = 5$. In x- bzw. y-Richtung gibt es jeweils $N = 4$ Punkte

Es werden nun alle Gitterpunkte des Gitters fortlaufend von links unten nach rechts oben (Abbildung 2.2.2) nummeriert. Stellt man nun Gleichung 2.14 für jeden Punkt auf, so erhält man eine $N^2 \times N^2$-Matrix. In dieser ist der 5-Punkt-Differenzenstern gut zu erkennen. Offensichtlich werden bei jedem Punkt des Gitters, der mindestens einen Randpunkt als Nachbarn hat, in der Matrix Werte unterdrückt. Diese tauchen später in der rechten Seite des Gleichungssystems wieder auf.

$$\mathbf{A} = \begin{pmatrix} A_1 & -Id & & \\ -Id & A_2 & \ddots & \\ & \ddots & \ddots & -Id \\ & & -Id & A_n \end{pmatrix}, \tag{2.16}$$

wobei $\mathbf{Id} \in \mathbb{R}^{N \times N}$ die Identität meint und für alle $i = 0, .., N$ gilt:

$$A_i = \begin{pmatrix} 4 & -1 & & & & & \\ -1 & 4 & -1 & & & & \\ & -1 & 4 & -1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -1 & 4 & -1 & \\ & & & & -1 & 4 & -1 \\ & & & & & -1 & 4 \end{pmatrix} \tag{2.17}$$

$A_i \in \mathbb{R}^{N \times N}$.

Unser Ziel ist ein lineares Gleichungssystem der Form $\mathbf{A}u = f$. Dafür muss zusätzlich die rechte Seite f aufgestellt werden. Da die Funktion $f(x, y)$ bekannt ist, spielen lediglich noch die Randpunkte eine wesentliche Rolle. Zu jeder Komponente von f, die einen Randpunkt als Nachbarn hat, wird dieser dazu addiert. Hat eine Komponente von f zwei Nachbarn am Rand von Ω_h, werden beide addiert. Dies führt uns auf folgende rechte Seite [Dahmen/Reusken]:

$$f = h^2 \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}, \tag{2.18}$$

wobei gilt

$$f_1 = \begin{pmatrix} f(h,h) + h^{-2}(g(h,0) + g(0,h)) \\ f(2h,h) + h^{-2}(g(2h,0)) \\ \vdots \\ f(1-2h,h) + h^{-2}(g(1-2h,0)) \\ f(1-h,h) + h^{-2}(g(1-h,0) + g(0,1-h)) \end{pmatrix}, \tag{2.19}$$

$$f_j = \begin{pmatrix} f(h, jh) + h^{-2}(g(0, jh)) \\ f(2h, jh) \\ \vdots \\ f(1-2h, jh) \\ f(1-h, jh) + h^{-2}(g(1, jh)) \end{pmatrix} 2 \leq j \leq N-1, \tag{2.20}$$

$$f_N = \begin{pmatrix} f(h, 1-h) + h^{-2}(g(h,1) + g(0, 1-h)) \\ f(2h, 1-h) + h^{-2}(g(2h, 1)) \\ \vdots \\ f(1-2h, 1-h) + h^{-2}(g(1-2h, 1)) \\ f(1-h, 1-h) + h^{-2}(g(1-h, 1) + g(1, 1-h)) \end{pmatrix}. \tag{2.21}$$

Schließlich erhalten wir das lineare Gleichungssystem der Form $\mathbf{A}u = f$, wobei $\mathbf{A}$ und f bekannt sind und u die approximierte Lösung, der partiellen Differentialgleichung darstellt.
Wir bezeichnet ab jetzt die diskrete 2D Poisson Matrix aus Gleichung 2.16 mit $\mathbf{A}_{2D}$.

2.3 Eigenschaften der Matrix $\mathbf{A}_{2D}$

Offensichtlich gilt $\mathbf{A}_{2D} = \mathbf{A}_{2D}^T$. Also ist $\mathbf{A}_{2D} \in \mathbb{R}^{n \times n}$ eine symmetrische Matrix. Somit existiert eine Orthogonalbasis aus Eigenvektoren für die gilt:

$$\mathbf{A}_{2D} v_{i,j} = \lambda_{i,j} v_{i,j}, \tag{2.22}$$

wobei $\lambda_{1,1}, ..., \lambda_{N,N} \in \mathbb{R}$ und $v_{i,j} \in \mathbb{R}^n$.

2.3.1 Eigenwerte und Eigenvektoren von $\mathbf{A}_{2D}$

Für $k \in \mathbb{N}$ und $h = \frac{1}{m}$ wie oben, seien $(x_k, y_l) \in \Omega_h$ mit $x_k := k \cdot h, y_l := l \cdot h$ und $\theta_k \in \mathbb{R}$ mit $\theta_k := k\pi h$. Dann gilt für die Eigenwerte:

$$\lambda_{k,l} = 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) \text{ für } 1 \leq k, l \leq N, \tag{2.23}$$

Für einen Eigenvektor am Punkt (x_k, y_l) gilt:

$$v_{k,l} = \sin(i\pi x_k)\sin(j\pi y_l) = \sin(i\theta_k)\sin(j\theta_l) \text{ für } 1 \leq i,j \leq N. \tag{2.24}$$

wobei N jeweils die Anzahl der Gitterpunkte in x- und in y-Richtung definiert.

Bemerkung:

Ein Eigenvektor im Punkt (x_k, y_l) lässt sich auch in der folgenden Form darstellen:

$$v_{k.l} = \begin{pmatrix} \sin(\theta_k)\sin(\theta_l) \\ \sin(\theta_k)\sin(2\theta_l) \\ \vdots \\ \sin(\theta_k)\sin(N\theta_l) \\ \sin(2\theta_k)\sin(\theta_l) \\ \sin(2\theta_k)\sin(2\theta_l) \\ \vdots \\ \sin(N\theta_k)\sin((N-1)\theta_l) \\ \sin(N\theta_k)\sin(N\theta_l) \end{pmatrix}. \tag{2.25}$$

Beispiel:

Um sich die Eigenvektoren besser vorstellen zu können, wollen wir zunächst drei Eigenvektoren des eindimensionalen Poisson Problems, auf das wir hier nicht näher eingehen möchten, betrachten. Das Gebiet $\Omega_h \in (0,1)$ mit einer Schrittweite $h = \frac{1}{15}$ und drei zugehörigen Eigenvektoren:

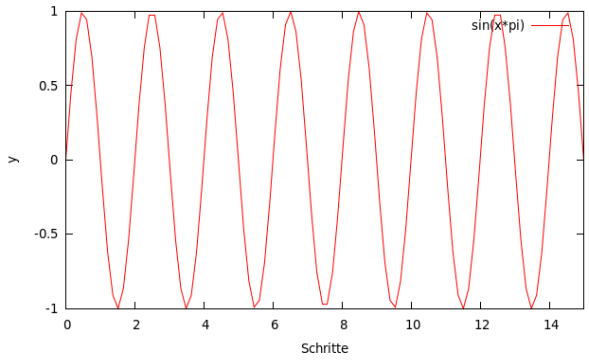

Abbildung 2.3: Dieser Graph stellt den Eigenwert am ersten Punkt, also $h = \frac{1}{15}$ dar. Zu sehen sind harmonische, kurzwellige Sinuskurven.

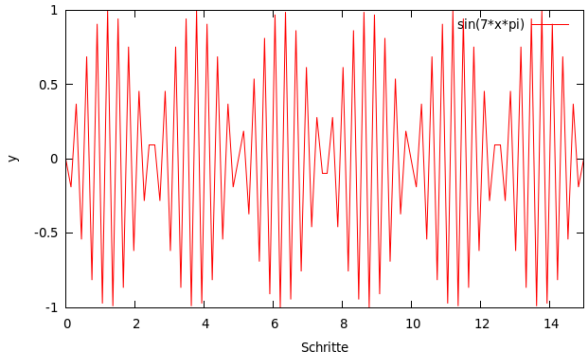

Abbildung 2.4: Bei diesen stark oszillierenden Sinuswellen handelt es sich um den Eigenvektor im Punkt $\frac{7}{15}$.

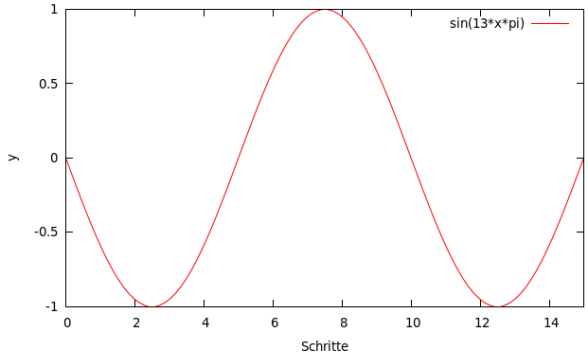

Abbildung 2.5: Diese gleichmäßigen langen Sinuswellen gehören zum Punkt $\frac{13}{15}$.

Man sieht deutlich, das die Oszillation unterschiedlich ist. Manche der Eigenvektoren sind langwellig, manche kurzwellig. Wenn wir nun zurück zu unserer Ausgangssituation des zweidimensionalen Poisson Problems gehen und das Gitter aus Abbildung 2.2.2 als Basis nehmen, erhalten wir für den Punkt 11 des Gitters folgende Darstellung des Eigenvektors:

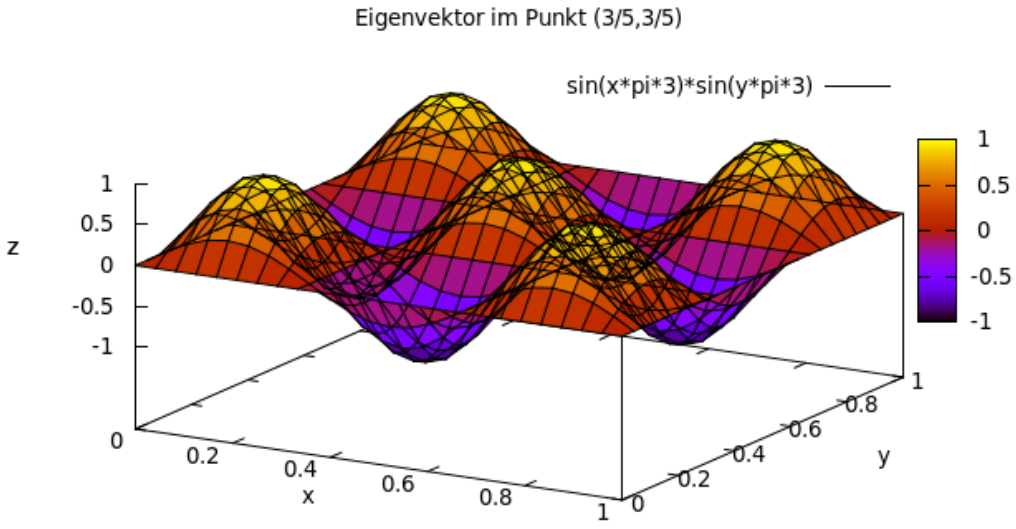

Abbildung 2.6: Man kann am Eigenvektor des Gitterpunktes $(\frac{3}{5}, \frac{3}{5})$ gut die gleichmäßigen Sinuswellen erkennen.

Auch hier sind die Sinuswellen gut zu erkennen. Man kann dieses Bild in etwa mit Abbildung 2.3.1 vergleichen. Dieser Eigenvektor besitzt eine langwellige Oszillation.

Beweis zu Abschnitt 2.3:

Wir wollen zunächst die $\mathbf{A}_i \in \mathbb{R}^{N\times N}$ von $\mathbf{A} \in \mathbb{R}^{n\times n}$ genauer Betrachten.
Behauptung: Für eine Matrix $\mathbf{B} \in \mathbb{R}^{N\times N}$ mit

$$\mathbf{B} = \begin{pmatrix} a & b & & & \\ c & a & b & & \\ & \ddots & \ddots & \ddots & \\ & & c & a & b \\ & & & c & a \end{pmatrix} \tag{2.26}$$

gilt für die Eigenwerte $\lambda_k := a + 2b\left(\frac{c}{b}\right)^{\frac{1}{2}}\cos(\theta_k)$ und die Eigenvektoren $v_{k_i} := \left(\frac{c}{b}\right)^{\frac{i}{2}}\sin(i\theta_k)$ für alle $1 \leq i \leq N, \lambda_k \in \mathbb{R}, v_k \in \mathbb{R}^{N\times N}$.

Beweis:

Da λ_k, v_k Eigenwerte bzw. Eigenvektoren von $\mathbf{B}$ sind, gilt folgende Gleichung:

$$(\mathbf{B} - \lambda_k \mathbf{Id}) v_k = 0 \tag{2.27}$$

$$\iff \begin{pmatrix} a-\lambda_k & b & & & \\ c & a-\lambda_k & b & & \\ & \ddots & \ddots & \ddots & \\ & & c & a-\lambda_k & b \\ & & & c & a-\lambda_k \end{pmatrix} v_k = 0 \tag{2.28}$$

$$\iff \begin{pmatrix} (a-\lambda_k)v_{k_1} + bv_{k_2} \\ cv_{k_1} + (a-\lambda_k)v_{k_2} + bv_{k_3} \\ \vdots \\ cv_{k_{N-2}} + (a-\lambda_k)v_{k_{N-1}} + bv_{k_N} \\ cv_{k_{N-1}} + (a-\lambda_k)v_{k_N} \end{pmatrix} = 0 \tag{2.29}$$

Wir wollen zunächst die einzelnen Summanden ausrechnen bzw. vereinfachen:

1. Löse $(a-\lambda_k)v_{k_i}$

$$\begin{aligned} (a-\lambda_k)v_{k_i} &= (a - (a + 2b\left(\frac{c}{b}\right)^{\frac{1}{2}}\cos(\theta_k)))\left(\frac{c}{b}\right)^{\frac{i}{2}}\sin(i\theta_k) \\ &= -2b\left(\frac{c}{b}\right)^{\frac{i+1}{2}}\cos(\theta_k)\sin(i\theta_k) \end{aligned}$$

2. Löse $bv_{k_{i+1}}$

$$\begin{aligned} bv_{k_{i+1}} &= b\left(\frac{c}{b}\right)^{\frac{i+1}{2}}\sin((i+1)\theta_k) \\ &= b\left(\frac{c}{b}\right)^{\frac{i+1}{2}}(\cos(\theta_k)\sin(i\theta_k) + \cos(i\theta_k)\sin(\theta_k)) \end{aligned}$$

3. Löse $cv_{k_{i-1}}$

$$\begin{aligned} cv_{k_{i-1}} &= c\left(\frac{c}{b}\right)^{\frac{i-1}{2}} \sin((i-1)\theta_k) = \left(c^2\frac{c}{b}\right)^{\frac{i-1}{2}} \sin((i-1)\theta_k) \\ &= \left(\frac{1}{b^{-2}}\frac{c}{b}\right)^{\frac{i+1}{2}} \sin((i-1)\theta_k) = b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} \sin((i-1)\theta_k) \\ &= b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} (\cos(\theta_k)\sin(i\theta_k) - \cos(i\theta_k)\sin(\theta_k)) \end{aligned}$$

Nun rechnen wir jede Zeile unseres Vektors aus:

1. Zeile 1, also $i = 1$

$$\begin{aligned} & -2b\left(\frac{c}{b}\right)^{\frac{1+1}{2}} \cos(\theta_k)\sin(\theta_k) + b\left(\frac{c}{b}\right)^{\frac{1+1}{2}} (\cos(\theta_k)\sin(\theta_k) + \cos(\theta_k)\sin(\theta_k)) \\ = \;& c(-2\cos(\theta_k)\sin(\theta_k) + 2\cos(\theta_k)\sin(\theta_k)) = 0. \end{aligned}$$

2. Zeile $2, .., N-1$, betrachte für alle $2 \leq i \leq N-1$

$$\begin{aligned} & b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} (\cos(\theta_k)\sin(i\theta_k) - \cos(i\theta_k)\sin(\theta_k)) - 2b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} \cos(\theta_k)\sin(i\theta_k) \\ + \;& b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} (\cos(\theta_k)\sin(i\theta_k) - \cos(i\theta_k)\sin(\theta_k)) \\ = \;& b\left(\frac{c}{b}\right)^{\frac{i+1}{2}} (-2\cos(\theta_k)\sin(i\theta_k) + 2\cos(\theta_k)\sin(i\theta_k) \\ + \;& \cos(i\theta_k)\sin(\theta_k) - \cos(i\theta_k)\sin(\theta_k)) = 0. \end{aligned}$$

3. Zeile N, somit für $i = N$

$$\begin{aligned} & b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} (\cos(\theta_k)\sin(N\theta_k) - \cos(N\theta_k)\sin(\theta_k)) - 2b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} \cos(\theta_k)\sin(N\theta \\ = \;& b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} (-2\cos(\theta_k)\sin(N\theta_k) + \cos(\theta_k)\sin(N\theta_k) - \cos(N\theta_k)\sin(\theta_k)) \\ = \;& -b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} (\cos(\theta_k)\sin(N\theta_k) + \cos(N\theta_k)\sin(\theta_k)). \\ = \;& -b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} \sin((N+1)\theta_k) \overset{h=\frac{1}{N+1}}{=} -b\left(\frac{c}{b}\right)^{\frac{N+1}{2}} \sin(k\pi) = 0. \end{aligned}$$

■

Für die Matrizen A_i erhalten wir mit $a = 4, b = c = -1$ für die Eigenwerte $\lambda_k = 4(1 - \frac{1}{2}\cos(\theta_k))$ und die Eigenvektoren $v_k = \sin(i\theta_k)$ für alle

$1 \leq i \leq N$.
Offensichtlich hat auch $-\mathbf{Id}$ die selben Eigenvektoren wie A_i, mit den Eigenwerte $\mu_k = -1$, denn

$$(-\mathbf{Id} - \lambda_k \mathbf{Id})v_k = \mathbf{0}v_k = 0. \tag{2.30}$$

Nun wollen wir diese Erkenntnisse für unsere Matrix $\mathbf{A}_{2D} \in \mathbb{R}^{n\times n}$ verwenden. Da diese Matrix ebenfalls eine Tridiagonalmatrix ist, folgt für $a = A_j$ und $b = c = -Id$.
Unser gesuchter Eigenvektor in einem Punkt (x_k, y_l) war gegeben durch $v_{k,l} = \sin(i\theta_k)\sin(j\theta_l) = \sin(j\theta_l)v_k$. Wegen $\mathbf{A}_{2D}$ symmetrisch folgt:

$$\mathbf{A}_{2D}v_{k,l} = \begin{pmatrix} A_1 & -Id & & & \\ -Id & A_2 & -Id & & \\ & \ddots & \ddots & \ddots & \\ & & -Id & A_{n-1} & -Id \\ & & & -Id & A_n \end{pmatrix} \begin{pmatrix} \sin(\theta_l)v_k \\ \sin(2\theta_l)v_k \\ \vdots \\ \sin((N-1)\theta_l)v_k \\ \sin(N\theta_l)v_k \end{pmatrix} = \lambda_{k,l} \begin{pmatrix} \sin(\theta_l)v_k \\ \sin(2\theta_l)v_k \\ \vdots \\ \sin((N-1)\theta \\ \sin(N\theta_l)v \end{pmatrix}$$

Nun wollen wir wie oben $\mathbf{A}_{2D}v_{k,l}$ explizit ausrechnen und müssen dafür wieder drei Fälle unterscheiden:

1. Zeile 1, also $j = 1$

$$\begin{aligned} &\implies A_1\sin(\theta_l)v_k - Id\sin(2\theta_l)v_k = \underbrace{A_1v_k}_{=\lambda_k v_k}\sin(\theta_l) + (\underbrace{-Idv_k}_{\mu_k v_k}) \underbrace{\sin(2\theta_l)}_{2\cos(\theta_l)\sin(\theta_l)} \\ = \;& \lambda_k v_k \sin(\theta_l) + 2\mu_k v_k \cos(\theta_l)\sin(\theta_l) = (\lambda_k + 2\mu_k\cos(\theta_l))\underbrace{v_k\sin(\theta_l)}_{v_{k,l}} \\ = \;& (\lambda_k + 2\mu_k\cos(\theta_l))v_{k,l} \overset{\text{einsetzen von } \lambda_k,\mu_k}{=} (4(1 - \frac{1}{2}\cos\theta_k) - 2\cos(\theta_l))v_{k,l} \\ = \;& (4 - 2\cos(\theta_k) - 2\cos(\theta_l))v_{k,l} = (2 - 2\cos(\theta_k) + 2 - 2\cos(\theta_l)) \\ = \;& (2(1 - \frac{1}{2}\cos(\theta_k)) + 2(1 - \frac{1}{2}\cos(\theta_l)) = 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) = \lambda_{k,l}v_{k,l} \end{aligned}$$

2. Zeile $2, .., N-1$, betrachte für alle $2 \leq j \leq N-1$

$$\begin{aligned}
&\Longrightarrow -Idv_k \sin((j-1)\theta_l) + A_j v_k \sin(j\theta_l) - Idv_k \sin((j+1)\theta_l) \\
= \;& \mu_k v_k(\sin(j\theta_l)\cos(\theta_l) - \cos(j\theta_l)\sin(\theta_l)) + \lambda_k v_k \sin(j\theta_l) \\
- \;& \mu_k v_k(\sin(j\theta_l)\cos(\theta_l) + \cos(j\theta_l)\sin(\theta_l)) \\
= \;& 2\mu_k v_k \sin(j\theta_l)\cos(\theta_l) + \lambda_k v_k \sin(j\theta_l) = (\lambda_k + \mu_k \cos(\theta_l))\sin(j\theta_l)v_k \\
= \;& (\lambda_k + \mu_k \cos(\theta_l))\sin(j\theta_l)v_{k,l} \overset{\text{wie oben}}{=} 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) = \lambda_{k,l}v_{k,l}
\end{aligned}$$

3. Zeile N, somit für $j = N$

$$\begin{aligned}
&\Longrightarrow -Idv_k \sin((N-1)\theta_l) + A_N v_k \sin(N\theta_l) \\
= \;& \mu_k v_k(\sin((N-1)\theta_l) + \underbrace{\sin((N+1)\theta_l)}_{=0}) + \lambda_k v_k \sin(N\theta_l) \\
= \;& \mu_k v_k(\sin(N\theta_l)\cos(\theta_l) - \cos(N\theta_l)\sin(\theta_l) + \sin(N\theta_l)\cos(\theta_l) \\
+ \;& \cos(N\theta_l)\sin(\theta_l)) + \lambda_k v_k \sin(N\theta_l) = 2\mu_k v_k \sin(N\theta_l)cos(\theta_l) + \lambda_k v_k \sin(N\theta_l) \\
= \;& (\lambda_k + 2\mu_k \cos(\theta_l))\sin(N\theta_l)v_k \overset{\text{wie oben}}{=} 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) = \lambda_{k,l}v_{k,l}
\end{aligned}$$

Also sind die $\lambda_{k,l}$ Eigenwerte von $\mathbf{A}_{2D}$ zu den Eigenvektoren $v_{k,l}$.

■

2.3.2 Folgerung ($\mathbf{A}_{2D}$ ist s.p.d.)

Für für die Eigenwerte gilt $\lambda_{k,l} \in \mathbb{R}_{>0}$, d.h. $\mathbf{A}_{2D}$ ist symmetrisch positiv definit.

Beweis:

Mit $\sin^2(x) \in (0,1)$ für $x \in (0, \frac{\pi}{2})$ und $0 < \frac{\pi h}{2} < \frac{\pi}{2}$ gilt:

$$\lambda_{k,l} = 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) > 0. \tag{2.31}$$

■

2.3.3 Definition (Kondition einer symmetrischen Matrix)

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

$$\kappa_2(A) := \frac{\lambda_{max}}{\lambda_{min}} \geq 1. \tag{2.32}$$

Je kleiner die Konditionszahl κ, desto besser ist eine Matrix konditioniert.

2.3.4 Lemma (Kondition von $\mathbf{A}_{2D}$)

Für die Kondition der Matrix $\mathbf{A}_{2D}$ gilt:

$$\kappa_2(A_{2D}) = \frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})}. \tag{2.33}$$

Beweis:

Es gilt mit $h = \frac{1}{m}$ und da $sin^2(x) \in (0,1)$ streng monoton steigend ist für $x \in (0, \frac{\pi}{2})$:

$$\lambda_{min} = \lambda_{1,1} = 8\sin^2(\frac{\pi h}{2}), \tag{2.34}$$

$$\begin{aligned}\lambda_{max} &= \lambda_{N,N} = 8\sin^2(\frac{N\pi h}{2}) \overset{\substack{h=\frac{1}{m},\\ N=m-1}}{=} 8\sin^2(\frac{(m-1)\pi}{2m}) \\ &= 8\sin^2(\frac{\pi}{2} - \frac{\pi}{2m}) = 8\sin^2(\frac{\pi}{2} - \frac{\pi h}{2}) \\ &= 8(\sin(\frac{\pi}{2}\cos(\frac{\pi h}{2})) - \cos(\frac{\pi h}{2})\sin(\frac{\pi}{2}))^2 = 8\cos^2(\frac{\pi h}{2})\end{aligned} \tag{2.35}$$

Somit folgt aus Gleichung 2.34 und Gleichung 2.35:

$$\kappa_2(A_{2D}) = \frac{\lambda_{N,N}}{\lambda_{1,1}} = \frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})}.$$

■

In [DahmenReusken] wird außerdem gezeigt, dass sich $\kappa_2(\mathbf{A}_{2D})$ wie folgt nähern lässt:

$$\frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} = \left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)). \tag{2.36}$$

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 von $\mathbf{A}_{2D}$.
Je größer m gewählt wird, also je feiner das Gitter wird, desto schlechter wird die Kondition der Matrix.

$$\left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)) = \left(\frac{2}{\pi \frac{1}{m}}\right)^2 (1 + \mathcal{O}(\frac{1}{m^2})) \approx \frac{4m^2}{\pi}. \tag{2.37}$$

Diesem Problem werden wir später nochmals gegenüber stehen.

3 Iterative Lösungsvefahren für lineare Gleichungssysteme

Gleichungssysteme, die partielle Differentialgleichungen lösen, können sehr groß werden. 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 zu langsam.
Wesentlich besser geeignet für diese Problemstellung sind iterative Verfahren. Sie zeichnen sich durch eine schnelle Konvergenz und einen geringeren Rechenaufwand aus.
Ein Großteil der Definitionen, Sätze und Lemmata in diesem Kapitel sind sinngemäß aus Dahmen/Reusken.

3.1 Grundbegriffe

3.1.1 Definition (Iterationsmatrix)

Sei $\mathbf{A} \in \mathbb{R}^{n\times n}$. Sei außerdem $\mathbf{C} \in \mathbb{R}^{n\times n}$ eine nichtsinguläre Matrix. Für die iterative Lösung eines linearen Gleichungssystems der Form $Au = f$ ist die Iterationsmatrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ definiert als:

$$\mathbf{T} := (\mathbf{Id} - \mathbf{CA}), \tag{3.1}$$

wobei die Iterationsvorschrift für $k = 1, ..., n$ gegeben ist durch:

$$u^{k+1} := (\mathbf{Id} - \mathbf{CA})u^k + \mathbf{C}f. \tag{3.2}$$

3.1.2 Definition (Spektralradius)

Sei $\mathbf{A} \in \mathbb{R}^{n \times n}$. Und seien für alle $i = 1, ..., n$, $\lambda_i \in \mathbb{R}$. Dann gilt:

$$\rho(A) := \max_{1 \leq i \leq n} |\lambda_i|. \tag{3.3}$$

Ist $\mathbf{A}$ symmetrisch so gilt auch $\rho(A) = \|A\|_2$ und $\lambda_i \in \mathbb{R}_{>0}$ für alle $i = 1, ..., n$.

3.1.3 Satz (Konvergenz iterativer Verfahren)

Ein iteratives Verfahren mit beliebigen Startvektor $x^0 \in \mathbb{R}^n$ konvergiert genau dann gegen die exakte Lösung $x^* \in \mathbb{R}^n$, wenn gilt:

$$\rho(\mathbf{T}) = \rho(Id - CA) < 1. \tag{3.4}$$

Einen Beweis hierzu findet man z.B. in (Dahmen/Reusken und Verweis).

3.1.4 Definition (Residuum und Fehler)

Sei $u^* \in \mathbb{R}^n$ die exakte Lösung des linearen Gleichungssystems $Au = f$. Sei außerdem $u^k \in \mathbb{R}^n$ die Approximation der Lösung im $k - ten$ Iterationsschritt. Dann gilt für das Residuum:

$$r^k := f - \mathbf{A}u^k. \tag{3.5}$$

Der Fehler, also die Diskrepanz zwischen exakter und approximierter Lösung, ist definiert als:

$$e^k := u^* - u^k. \tag{3.6}$$

Durch Multiplikation mit der Matrix $\mathbf{A}$ ergibt sich:

$$e = u^* - u \Leftrightarrow \mathbf{A}e = \mathbf{A}(u^* - u) \Leftrightarrow \mathbf{A}e = \mathbf{A}u^* - \mathbf{A}u \Leftrightarrow \mathbf{A}e = b - \mathbf{A}u \Leftrightarrow \mathbf{A}e = r. \tag{3.7}$$

$\mathbf{A}e = r$ nennen wir Residuumsgleichung.

3.2 Das Jacobi-Verfahren (Gesamtschrittverfahren)

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

$$A = D - L - U. \tag{3.8}$$

Dabei sind $\mathbf{D}, \mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times n}$, wobei $\mathbf{D} = diag(a_{1,1}, ..., a_{n,n})$ und $\mathbf{L}$ eine strikte untere und $\mathbf{U}$ eine strikte obere Dreiecksmatrix ist.
Somit ergibt sich für $\mathbf{A}u = f$:

$$Au = f \Leftrightarrow (D - L - U)u = f \Leftrightarrow Du = (L + U)u + f. \tag{3.9}$$

Ist nun $\mathbf{D}$ nicht singulär, so gilt für das Jacobi-Verfahren folgende Iterationsvorschrift:

$$Du^{k+1} = (L + U)u^k + f \Leftrightarrow u^{k+1} = D^{-1}(L + U)u^k + D^{-1}f. \tag{3.10}$$

3.2.1 Algorithmus (Jacobi-Verfahren)

In Komponentenschreibweise mit einem Startvektor $u^0 \in \mathbb{R}^n$ *beliebig* und $k = 1, 2, ...$ Berechne für $i = 1, ..., n$:

$$u_i^{k+1} = \frac{1}{a_{ii}}(f_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}u_i^k). \tag{3.11}$$

In jedem Schritt zur Berechnung von u^{k+1} muss hier die Information seines Vorgängers u^k bekannt sein. Der Rechenaufwand pro Iterationsschritt beträgt $\mathcal{O}(n^2)$ und entspricht somit einer Matrix-Vektor-Multiplikation.

3.2.2 Satz (Iterationsmatrix des Jacobi-Verfahrens)

Für die Iterationsmatrix des Jacobi-Verfahrens gilt:

$$\mathbf{T}_J := (\mathbf{Id} - \mathbf{D}^{-1}\mathbf{A}) \tag{3.12}$$

Hier ist also $\mathbf{C} = \mathbf{D}^{-1}$

Beweis:

Mit der Iterationsvorschrift folgt:

$$\begin{aligned} u^{k+1} &= D^{-1}(L+U)u^k + D^{-1}f \overset{(L+U)=(D-A)}{=} D^{-1}(D-A)u^k + D^{-1}f \\ &= (Id - D^{-1}A)u^k + D^{-1}f. \end{aligned}$$

Also $\mathbf{T}_J = (\mathbf{Id} - \mathbf{D}^{-1}\mathbf{A})$.

■

3.2.3 Satz (Eigenwerte der Jacobi-Iterationsmatrix bzgl. $\mathbf{A}_{2D}$)

Man sieht leicht ein, dass die Eigenvektoren von $\mathbf{T}_J$ gleich denen von $\mathbf{A}_{2D}$ sind. Dann gilt für die Eigenwerte der Iterationsmatrix:

$$\lambda_{i,j}(\mathbf{T}_J) = 1 - \left(\sin^2(\frac{\theta_i}{2}) + \sin^2(\frac{\theta_j}{2})\right). \tag{3.13}$$

für $1 \leq i, j \leq N$ und θ_i, θ_j wie in Unterabschnitt 2.3.1.

Beweis:

Dieser folgt direkt mit dem Beweis aus Unterabschnitt 3.3.2 für $\omega = 1$.

■

3.2.4 Lemma (Spektralradius der Jacobi-Iterationsmatrix bzgl. $\mathbf{A}_{2D}$)

Das Jacobi-Verfahren konvergiert für die diskrete Poisson Gleichung und es gilt für den Spektralradius:

$$\rho(\mathbf{T}_J) = \cos(\pi h) < 1. \tag{3.14}$$

Beweis:

Folgt mit Beweis aus Unterabschnitt 3.3.3 und $\omega = 1$.

■

Das Jacobi-Verfahren konvergiert also für $\mathbf{A}_{2D}$. Allerdings nimmt mit feinerem Gitter, also kleinere Schrittweite h, die Konvergenzgeschwindigkeit stark ab, da der Spektralradius nahe bei 1 liegt. Sie dazu auch Unterabschnitt 3.3.3 Abschließend wollen wir den Rechenaufwand verbessern. Dafür nutzen wir die Dünnbesetztheit von $\mathbf{A}_{2D}$ aus. Es sind pro Zeile maximal 5 Einträge ungleich Null. Oder anders formuliert, hat jeder Gitterpunkt in Ω_h höchstens 4 Nachbarn. Nutzt man diese Struktur aus, so erhält man den folgenden Algorithmus.

3.2.5 Algorithmus (Jacobi-Verfahren für $\mathbf{A}_{2D}$) [NumParVer]

Berechne für $k = 1, 2, ...$ mit Startvektor $u^0 \in \mathbb{R}^n$ *beliebig*
Für $i = 1, ..., N$ und für $j = 1, ..., N$:

$$u_{i,j}^{k+1} = \frac{1}{a_{i,i}}(f_{i,j} - u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k). \tag{3.15}$$

(Werte, bei denen eine Null im Index steht, werden ignoriert!)
Der Rechenaufwand pro Iterationsschritt beträgt lediglich $\mathcal{O}(N \cdot N) = \mathcal{O}(n)$ Schritte.

3.3 Das Jacobi-Relaxationsverfahren

Wir wollen nochmal das Jacobi-Verfahren betrachten:

$$u^{k+1} = (Id - D^{-1}A)u^k + D^{-1}f. \tag{3.16}$$

Zunächst soll eine Umformung erfolgen:

$$\begin{aligned} u^{k+1} &= (Id - D^{-1}A)u^k + D^{-1}f \\ &= u^k - D^{-1}Au^k + D^{-1}f \\ &= u^k + D^{-1}\underbrace{(f - Au^k)}_{=r^k}. \end{aligned} \tag{3.17}$$

Wir addieren also zu u^k das Residuum. Die Idee ist nun dieses mit einem Parameter $\omega \in \mathbb{R}$ zu multiplizieren:

$$\begin{aligned} u^{k+1} &= u^k + D^{-1}\omega r^k = u^k + \omega D^{-1}(f - Au^k) \\ &= u^k - \omega D^{-1}Au^k + \omega D^{-1}f = (Id - \omega D^{-1}A)u^k + \omega D^{-1}f \\ &= (Id - \omega Id + \omega Id - \omega D^{-1}A)u^k + \omega D^{-1}f \\ &= (1-\omega)Id + \omega(Id - D^{-1}A)u^k + \omega D^{-1}f. \end{aligned} \tag{3.18}$$

Gleichung 3.18 stellt die Iterationsvorschrift für das Jacobi-Relaxationsverfahren. Die Iterationsmatrix ist offensichtlich gegeben durch:

$$\mathbf{T}_{J_\omega} := (1-\omega)\mathbf{Id} + \omega(\mathbf{Id} - \mathbf{D}^{-1}\mathbf{A}) = (\mathbf{Id} - \omega\mathbf{D}^{-1}\mathbf{A}). \tag{3.19}$$

Letztlich erweitert man den Jacobi-Algorithmus also mit dem Parameter ω und addiert $(1-\omega)u^k$ zu u^{k+1}.

3.3.1 Algorithmus (Jacobi-Relaxations-Verfahren)

Sei $u^0 \in \mathbb{R}^n$ ein *beliebiger* Startvektor und $k = 1, 2, ...$ berechne für $i = 1, ..., n$:

$$u_i^{k+1} = (1-\omega)u_i^k + \frac{\omega}{a_{ii}}(f_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}u_i^k). \tag{3.20}$$

Beachte: Für $\omega = 1$ erhalten wir das Jacobi-Verfahren. Auch hier beträgt der Rechenaufwand $\mathcal{O}(n^2)$.

3.3.2 Satz (Eigenwerte des Jacobi-Relaxationsverfahren bzgl. $\mathbf{A}_{2D}$)

Die Eigenvektoren entsprechen denen von $\mathbf{A}_{2D}$ und für die Eigenwerte von $\mathbf{T}_{J_\omega}$ gilt:

$$\lambda_{i,j}(\mathbf{T}_J) = 1 - \omega\left(\sin^2(\frac{\theta_i}{2}) + \sin^2(\frac{\theta_j}{2})\right) \text{ für } 1 \leq i, j \leq N. \tag{3.21}$$

θ_i, θ_j wie in Unterabschnitt 2.3.1.

Beweis:

Für $\mathbf{D} = 4\mathbf{Id}$ folgt $\mathbf{D}^{-1} = \frac{1}{4}\mathbf{Id}$. Für die Iterationsmatrix $\mathbf{T}_{J_\omega}$ angewandt auf einen Vektor u gilt:

$$\mathbf{T}_{J_\omega}u = (\mathbf{I}d - \omega\mathbf{D}^{-1}\mathbf{A})u = \mathbf{Id}u - \omega\mathbf{D}^{-1}\underbrace{\mathbf{A}u}_{=\lambda_{i,j}(A)u} = \mathbf{Id}u - \frac{\omega}{4}\mathbf{IdA}u = u(1 - \frac{\omega}{4}\lambda_{i,j}(\mathbf{A})).$$

Die Eigenwerte von $\mathbf{T}_{J_\omega}$ lassen sich also einfach durch $(1 - \frac{\omega}{4}\lambda_{i,j}(A))$ berechnen:

$$\lambda_{i,j}(\mathbf{T}_J) = 1 - \omega\left(\sin^2(\frac{\theta_i}{2}) + \sin^2(\frac{\theta_j}{2})\right) \text{ für } 1 \leq i,j \leq N.$$

■

3.3.3 Lemma (Spektralradius der Jacobi-Relaxations-Matrix bzgl. $\mathbf{A}_{2D}$)

Das Jacobi-Relaxations-Verfahren konvergiert für die diskrete Poisson Gleichung und es gilt für den Spektralradius:

$$\rho(\mathbf{T}_{J_\omega}) = 1 - \omega(1 - \cos(\pi h)) < 1. \tag{3.22}$$

Beweis:

Mit Gleichung 3.21 folgt:

$$\begin{aligned}
\rho(\mathbf{Id} - \omega\mathbf{D}^{-1}\mathbf{A}_{2D}) &= \max_{1\leq i,j\leq N}|\lambda_{i,j}| = \max_{1\leq i,j\leq N}|1 - \omega\left(\sin^2(\frac{\theta_i}{2}) + \sin^2(\frac{\theta_j}{2})\right)| \\
&= 1 - 2\omega\sin^2(\frac{\theta_1}{2}) = 1 - 2\omega\sin^2(\frac{\pi h}{2}) = 1 - 2\omega(\frac{1}{2}(1 - \cos(\pi h)) \\
&= 1 - \omega(1 - \cos(\pi h)) \overset{\text{Taylorformel}}{\approx} 1 - \omega(1 - (1 - \frac{\pi^2 h^2}{2})) \\
&= 1 - \frac{\omega}{2}\pi^2 h^2 < 1.
\end{aligned}$$

■

Die Iterationsmatrix des Jacobi-Relaxationsverfahrens bezüglich für $\mathbf{A}_{2D}$ und für ein $\omega \in (0,1)$ eine bessere Kondition, als die des Standard-Jacobi-Verfahrens. Häufig verwendete Werte im zweidimensionalen Fall sind $\omega = \frac{1}{2}$ und $\omega = \frac{4}{5}$, wobei letzterer Wert für das Mehrgitterverfahren den optimalen Parameter darstellt [Saad]. Leider ist für kleines h die Konvergenz immer noch sehr langsam. Den Rechenaufwand kann man allerdings auch hier verbesseren. Mit der selben Vorgehensweise wie in Unterabschnitt 3.2.5 erhalten wir:

3.3.4 Algorithmus (Jacobi-Relaxations-Verfahren für $\mathbf{A}_{2D}$) [NumParVer]

Berechne für $k = 1, 2, ...$ mit Startvektor $u^0 \in \mathbb{R}^n$ *beliebig*
Für $i = 1, ..., N$ und für $j = 1, ..., N$:

$$u_{i,j}^{k+1} = (1-\omega)u^k + \frac{\omega}{a_{i,i}}(f_{i,j} - u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k), \tag{3.23}$$

mit Rechenaufwand $\mathcal{O}(n)$.

3.4 Glättungseigenschaft

Für die Iterationsmatrix des Jacobi-Relaxationsverfahrens gilt:

$$u^{k+1} = (Id - \omega D^{-1}A)u^k + \omega D^{-1}f. \tag{3.24}$$

Die Eigenvektoren sind gegeben durch:

$$v_{k,l} = sin(i\theta_k)sin(j\theta_l) \text{ für } 1 \leq i, j, k, l \leq N \tag{3.25}$$

für θ_k, θ_l wie oben.
Als Eigenvektoren der Matrizen $\mathbf{T}_J$ bzw. $\mathbf{T}_{J_\omega}$ bilden diese Vektoren eine Basis des $\mathbb{R}^n$, wobei $n = N^2$. Für den Fehlerterm im $k-ten$ Iterationsschritt gilt:

$$e^k = u^* - u^k. \tag{3.26}$$

Betrachten wir nun den $(k+1) - ten$ Fehler und formen geschickt um, so gilt:

$$\begin{aligned} e^{k+1} &= u^* - u^{k+1} = u^* - \Big((Id - \omega D^{-1}A)u^k + \omega D^{-1}f\Big) \\ &= \underbrace{u^* - u^k}_{=e^k} + \omega D^{-1}Au^k - D^{-1}f = e^k + \omega D^{-1}(Au^k - f) \\ &= e^k + \omega D^{-1}(Au^k - Au^*) = e^k + \omega D^{-1}A(u^k - u^*) \\ &= e^k + \omega D^{-1}Ae^k = (Id - \omega D^{-1}A)e^k. \end{aligned} \tag{3.27}$$

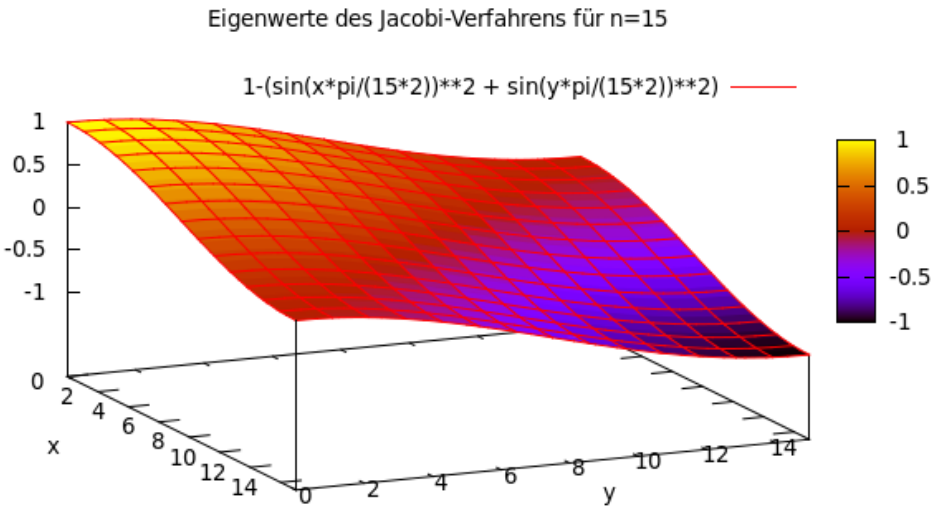

Abbildung 3.1: Dies ist das Eigenwertspektrum für $\omega = 1$. Das Gesamtschrittverfahren hat keine Glättungseigenschaft. Es liegen nur wenige der Eigenwerte um die Null.

Da die n Eigenvektoren $v_{i,j}$ eine Basis bilden, lässt sich der Fehler e als

Linearkombination der $v_{i,j}$ darstellen:

$$
\begin{aligned}
e^{k+1} &= \sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k+1} v_{i,j} = (Id - \omega D^{-1} A) \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} v_{i,j} \right) \\
&= \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} (Id - \omega D^{-1} A) v_{i,j} \right) = \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} (v_{i,j} - \frac{\omega}{4} \underbrace{A v_{i,j}}_{=\lambda_{i,j}(A) v_{i,j}}) \right) \\
&= \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} \underbrace{(1 - \frac{\omega}{4} \lambda_{i,j}(A))}_{=\lambda_{i,j}(\mathbf{T}_{J_\omega})} v_{i,j} \right) = \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} \lambda_{i,j}(\mathbf{T}_{J_\omega}) v_{i,j} \right) . \quad (3.28)
\end{aligned}
$$

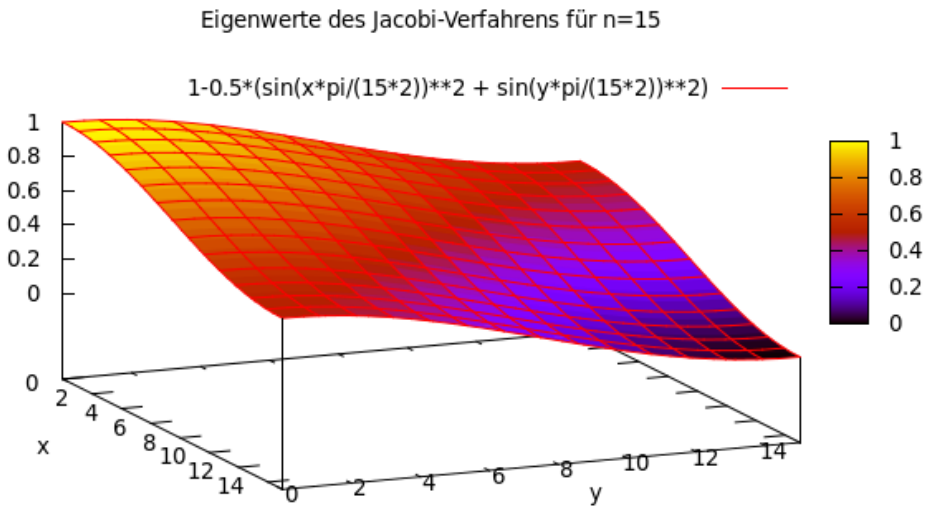

Abbildung 3.2: Für $\omega = \frac{1}{2}$ ist gut zu erkennen, dass für i, j nahe N viele Eigenwerte nahe Null liegen.

Betrachten wir nun die Eigenwerte der Iterationsmatrix für $\omega = 1$ (Jacobi-Verfahren), sieht man, dass die Eigenwerte für i, j nahe Null oder i, j nahe N den Fehler schlecht bis gar nicht dämpfen (Abbildung 3.1). Interessanter weiße ist der optimale Parameter $\omega = 1$, wenn man das Ganze als iteratives Verfahren verwendet [Saad]. Das Jacobi-Verfahren besitzt aber offensichtlich keine Glättungseigenschaft.

Da wir eine Fehlerglättung erreichen wollen, wählen wir nun $\omega = \frac{1}{2}$ (Abbildung 3.2). Wir stellen fest, dass die Eigenwerte der Iterationsmatrix zwischen 0 und 1 liegen. Sind $i, j > \frac{N}{2}$ so werden die Fehleranteile wesentlich besser gedämpft, da die Eigenwerte hier nahe Null liegen. Diese Eigenschaft nennt man die *Glättungseigenschaft*.
Es stellt sich z.B. heraus [Saad], dass der optimale Relaxationsparameter, der unabhängig von der Schrittweite h gewählt werden kann, $\omega = \frac{4}{5}$ ist. In Abbildung 3.3 ist gut zu sehen, dass ein Großteil des Spektrums nahe Null liegt.

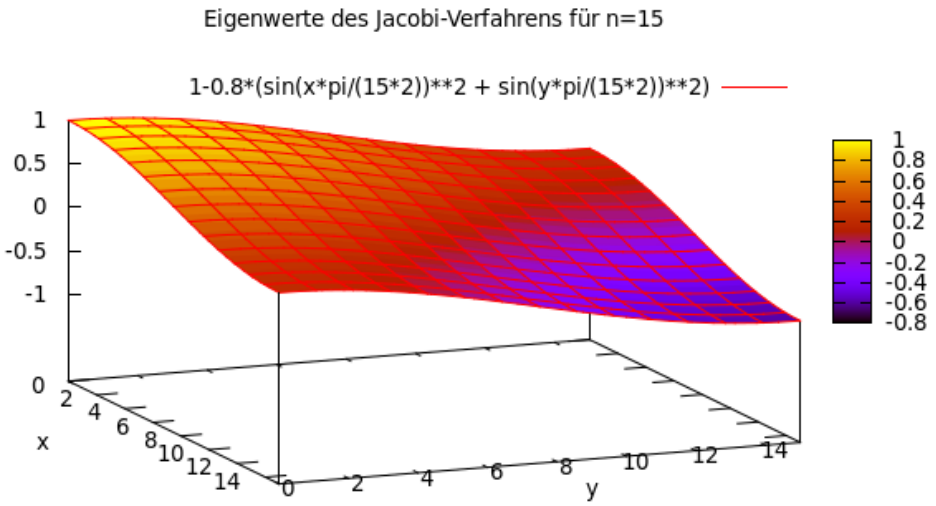

Abbildung 3.3: i
st gut zu erkennen, dass viele Eigenwerte um die Null liegen und somit eine gute Glättungseigenschaft besteht.] Für den optimalen Parameter $\omega = \frac{4}{5}$ [Saad] ist gut zu erkennen, dass viele Eigenwerte um die Null liegen und somit eine gute Glättungseigenschaft besteht.

3.5 Das Verfahren der konjugierten Gradienten

Das Verfahren der konjugierten Gradienten wurde 1952 von Hestenes und Stiefel erstmals vorgestellt. Es zeichnet sich durch Stabilität und schnelle Konvergenz aus.

Das CG-Verfahren (conjugate gradient) - wie es auch genannt wird - ist eine Projektions- und Krylow-Raum-Methode.

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 $x^T Ay = 0$ gilt.

3.5.2 Satz

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

$$f(u) := \frac{1}{2} u^T Au - f^T u, \tag{3.29}$$

wobei $f, u \in \mathbb{R}^n$. Dann gilt:

f hat ein eindeutig bestimmtes Minimum und

$$Au^* = f \Longleftrightarrow f(u^*) = \min_{u \in \mathbb{R}^n} f(u) \tag{3.30}$$

Einen Beweis hierzu findet man z.B. in [Dahmen/Reusken].
Es ist also äquivalent die Funktion $f(u)$ zu minimieren und das Gleichungssystem $Au = f$ zu lösen. Betrachtet man nun den Gradienten von $f(x)$, so stellt man fest, dass gilt:

$$\nabla f(x) = Ax - f = -r. \tag{3.31}$$

Da wir also stets das Minimum im Teilraum U_k suchen, wird uns folgendes Lemma hilfreich sein:

3.5.3 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 $u^k \in U_k$:

$$\|u^k - v\|_A = \min_{u \in U_k} \|u - v\|_A \tag{3.32}$$

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

$$P_{U_k,\langle\cdot,\cdot\rangle}(v) = \mathbf{u}^k = \sum_{j=0}^{k-1} \frac{\langle v, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \tag{3.33}$$

Der Beweis zu diesem Lemma folgt direkt aus dem Projektionssatz [Dahmen/Reusken]

3.5.4 Allgemeiner Algorithmus der konjugierten Gradienten

Zur Erzeugung der Lösung von u^* durch Näherungen $u^1, u^2, ...$ definieren wir folgende Teilschritte [Dahmen/Reusken]:

0. Definiere Teilraum U_1 und bestimme r^0 mit beliebigen Startvektor u^0

$$U_1 := span\{r^0\}, \text{ wobei } r^0 = f - Au^0 \tag{3.34}$$

1. Bestimme eine A-orthogonale Basis

$$p^0, ..., p^{k-1} \text{ von } U_k. \tag{3.35}$$

2. Bestimme eine Näherungslösung u^k, so dass gilt:

$$\|u^k - u^*\|_A = \min_{u \in U_k} \|x - u^*\|_A. \tag{3.36}$$

Wir berechnen also:

$$u^k = \sum_{j=0}^{k-1} \frac{\langle u^*, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j. \tag{3.37}$$

3. Erweitere den Teilraum U_k und berechne erneut das Residuum

$$U_{k+1} := span\{p^0, ..., p^{k-1}, r^k\} \text{ wobei } r^k := f - Au^k. \tag{3.38}$$

Nachdem man ein Residuum berechnet hat, startet der erste Iterationsschritt: Man erweitert seinen Teilraum um das Residuum und bestimmt darauf hin eine A-orthogonale Basis dieses Teilraumes. Ein gängiges Verfahren ist das Gram-Schmidt-Orthonormalisierungsverfahren. Die neue Näherungslösung bzgl. U_k kann dann über den (A-orthogonalen) Projektionssatz bestimmt werden. Nachdem erneut ein Residuum berechnet wurde, startet der nächste Iterationsschritt.
Wegen Gleichung 3.37 könnte man vermuten, dass u^* zur Durchführung des Algorithmus bekannt sein muss. Die folgenden Lemmata werden zeigen, dass dem nicht so ist.

3.5.5 Lemma

Sei $u^* \in \mathbb{R}^n$ die Lösung von $Au = f$. Dann gilt für ein $y \in U_k$:

$$\langle u^*, y \rangle_A = \langle f, y \rangle \tag{3.39}$$

Beweis:

Wir nutzen die Eigenschaften des A-Skalarproduktes aus:

$$\langle u^*, y \rangle_A = {u^*}^T Ay = y^T Au^* = y^T f = f^T y = \langle f, y \rangle.$$

■

Nun wollen wir Gleichung 3.37 neu formulieren.

3.5.6 Lemma

Sei $u^* \in \mathbb{R}^n$ die Lösung von $Au = f$ und $u^k \in \mathbb{R}^n$ die optimale Approximation von u^* in U_k. Dann kann u^k wie folgt berechnet werden:

$$u^k = u^{k-1} + \alpha_{k-1} p^{k-1}, \text{ mit } \alpha_{k-1} := \frac{\langle r^0, p^{k-1} \rangle}{\langle p^{k-1}, Ap^{k-1} \rangle}.$$

Einen Beweis findet man in [Dahmen/Reusken].

Bemerkung: u^k kann dadurch mit wenig Aufwand aus u^{k-1} und p^{k-1} berechnet werden.

3.5.7 Lemma

Das Residuum $r^k \in \mathbb{R}^n$ kann einfach berechnet werden durch:

$$r^k = r^{k-1} - \alpha_{k-1} A p^{k-1}, \tag{3.40}$$

wobei α_{k-1} wie in Gleichung 3.40.

Beweis:

$$\begin{aligned} & u^k = u^{k-1} + \alpha_{k-1} p^{k-1} \\ \iff & Au^k = Au^{k-1} + \alpha_{k-1} A p^{k-1} \\ \iff & b - Au^k = b - Au^{k-1} - \alpha_{k-1} A p^{k-1} \\ \implies & r^k = r^{k-1} - \alpha_{k-1} A p^{k-1}. \end{aligned}$$

■

Da wir nun u^k und r^k recht komfortabel bestimmen können, wollen wir im Folgenden noch eine Möglichkeit sehen, um p^k schnell zu berechnen.

3.5.8 Satz (Bestimmung einer A-orthogonalen Basis)

Durch

$$p^{k-1} = r^{k-1} - \sum_{j=0}^{k-2} \frac{\langle r^{k-1}, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j, \tag{3.41}$$

wird die A-orthogonale Basis zum Vektor r^{k-1} bestimmt, wobei $p^{k-1}, r^{k-1} \in \mathbb{R}^n$.

Beweis:

Der Beweis folgt direkt aus dem Gram-Schmidt-Orthonormalisierungsverfahren, welches allerdings einen hohen Rechenaufwand vorweist.
Wir wollen ohne Beweis (siehe z.B. [Dahmen/Reusken]) angeben, wie man die p^k effizienter bestimmen kann.

3.5.9 Satz

Für die Berechnung von p^k gilt:

$$p^{k-1} = r^{k-1} - \frac{\langle r^{k-1}, Ap^{k-2}\rangle}{\langle p^{k-2}, Ap^{k-2}\rangle} p^{k-2}. \tag{3.42}$$

Substituiert man nun geschickt einige Werte in den Skalarprodukten, führt das auf den Algorithmus der konjugierten Gradienten.

3.5.10 Numerischer Algorithmus der konjugierten Gradienten

Gegeben ist eine symmetrisch positiv definite Matrix $\mathbf{A} \in \mathbb{R}^n$. Bestimme die (Näherungs-) Lösung u^* mit Hilfe eines *beliebigen* Startvektors $u^0 \in \mathbb{R}^n$ zu einer gegebenen rechten Seite $b \in \mathbb{R}^n$. Setze $\beta_{-1} := 0$ und berechne das Residuum $r^0 = b - Au^0$.
Für $k = 1, 2, ...$, falls $r^{k-1} \neq 0$ berechne:

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

Man muss in diesem Algorithmus pro Iterationsschritt lediglich zwei Skalarprodukte ausrechnen (die r^{k-1}-Skalarprodukte können für die r^{k-2} nach der Berechnung des neuen Residuums wieder verwendet werden!) und eine Matrix-Vektor-Multiplikation durchführen. Somit erhält man einen Rechenaufwand von $\mathcal{O}(n^2)$. Angewandt auf $\mathbf{A}_{2D}$ kann man - durch das Ausnutzen der Dünnbesetztheit - einen Aufwand pro Schritt von $\mathcal{O}(n)$ erreichen. Dies erreicht man durch eine geschickt gewählte Matrix-Vektor-Multiplikation, die die Struktur von $\mathbf{A}_{2D}$ ausnutzt.

An dieser Stelle soll noch ein kurzer Satz über die Konvergenz des Verfahrens folgen. Einen Beweis hierzu findet man z.B. in (Dahmen/Reusken 573):

3.5.11 Satz (Konvergenz des CG-Algorithmus) [DahmenReusken]

Sei $\mathbf{A} \in \mathbb{R}^{n \times n}$ symmetrisch, positiv definit und seien $u, f, u^*, u^k \in \mathbb{R}^n$, wobei u^* die exakte Lösung des Gleichungssystems $Au = b$ und u^k die approximierte Lösung durch das CG-Verfahren ist. Dann gilt für $k = 1, 2, ...$:

$$\|u^k - u^*\|_A \leq 2 \left(\frac{\kappa_2(A) - 1}{\kappa_2(A) + 1} \right)^k \|u^0 - u^*\|_A \tag{3.43}$$

Da stets $\frac{\kappa_2(A)-1}{\kappa_2(A)+1} < 1$ gilt, sichert dieser Satz die Konvergenz des Algorithmus. Man sieht also, dass die Konvergenz des Verfahrens von der Kondition der Matrix $\mathbf{A}_{2D}$ abhängt.

Bemerkung:

Das Verfahren der konjugierten Gradienten konvergiert im Allgemeinen nicht, wenn **A** nicht symmetrisch, positiv definit ist.

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 2.3.4 gesehen haben, ist $\mathbf{A}_{2D}$ für feinere Gitter schlecht konditioniert. Und in Unterabschnitt 3.5.11 haben wir gesehen, dass die Konvergenz des CG-Verfahrens von eben dieser Matrix abhängt. Dies mindert natürlich die Effizienz des Verfahrens. Die Idee ist nun, die bei der Iteration zu Grunde liegende Matrix **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:

$$\mathbf{A}u = f \iff \mathbf{W}^{-1}\mathbf{A}u = \mathbf{W}^{-1}f. \tag{3.44}$$

Es macht also keinen Unterschied, ob wir $Au = f$ oder das äquivalente System lösen.

3.6.1.1 Beweis:

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

■

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 klein wird. Nun ist im Allgemeinen $\mathbf{W}^{-1}\mathbf{A}$ nicht s.p.d. Somit könnten wir zwar den CG-Alorithmus trotzdem anwenden, werden aber wegen dieser Tatsache möglicherweise keine Konvergenz erhalten. Um dies zu umgehen findet man einen Lösungsansatz (z.B. in Dahmen/Reusken 576), 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 u^* mit Hilfe eines *beliebigen* Startvektors $u^0 \in \mathbb{R}^n$ zu einer gegebenen rechten Seite $b \in \mathbb{R}^n$. Setze $\beta_{-1} := 0$, berechne das Residuum $r^0 = b - Au^0$ und $z^0 = \mathbf{W}^{-1}r^0$ (löse $\mathbf{W}z^0 = r^0$).
Für $k = 1, 2, ...$, falls $r^{k-1} \neq 0$ berechne:

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

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 **A**, die symmetrisch positiv definit ist, lässt sich durch eine Cholesky-Zerlegung in eine normierte untere Dreiecksmatrix **L** und eine rechte obere Dreicksmatrix **U** zerlegen, wobei gilt: $\mathbf{U} := \mathbf{D}\mathbf{L}^T$
Mit dieser Zerlegung möchten wir nun unser System vorkonditionieren. Allerdings würde eine vollständige Cholesky-Zerlegung viele Nulleinträge in einer dünn besetzten Matrix auslöschen. Darum greift man auf eine unvollständige Cholesky-Zerlegung zurück, bei der die Stellen, an denen **A** Nulleinträge besitzt, in L und U ebenfalls Null bleiben.

3.6.3.1 Definition (Muster E)

Sei $E \subseteq \{1, ..., n\} \times \{1, ..., n\}$ ein Muster, für das gilt:

$$E := \{(i, j) | 1 \leq i, j \leq n, a_{i,j} \neq 0\}. \tag{3.45}$$

Dann lässt sich die Matrix **A** auch folgendermaßen schreiben:

$$\mathbf{A} = \approx \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T. \tag{3.46}$$

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 Dreicksmatrix
- 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 Algorithmus effizient und den Rechenaufwand

so klein wie möglich zu machen, werden Summen nur über Indizes aus dem Muster berechnet.

3.6.3.3 Der numerische Algorithmus der unvollständigen Cholesky Zerlegung

Seien $\mathbf{A} \in \mathbb{R}^{n \times n}$ und E das Muster zur Matrix $\mathbf{A}$. Setze $\mathbf{L} = \mathbf{Id}$, $\mathbf{R} = 0$. Berechne dann für $i = 1, 2, ..., n$:

$$l_{i,i} = \left(a_{i,i} - \sum_{j=1,(i,j)\in E}^{i-1} l_{i,j}^2 \right)^{\frac{1}{2}} \tag{3.47}$$

$$\text{for k} = i+1, ..., n : \text{ if } (k,i) \in E : \tag{3.48}$$

$$l_{k,i} = \left(a_{k,i} - \sum_{j=1,(k,j)\in E,(i,j)\in E}^{k-1} l_{k,j} l_{i,j} \right) / l_{i,i} \tag{3.49}$$

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. Rückwärtseinsetzen lösbar.
- Für viele Probleme zeigt sich, dass $\kappa_2(\mathbf{W}^{-1}\,\mathbf{A}\,) \ll \kappa_2(A)$ gilt.

Es gibt einige Varianten dieses Verfahrens. Wir wollen uns an dieser Stelle noch mit einer dieser 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 $e := (1, , 1, ..., 1)^T$

- $a_{i,j} = (\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T)_{i,j}$ für alle $(i,j) \in E, i \neq j$
- $\mathbf{A}e = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T e$, 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 $i = 1, 2, ..., n$:

$$
\begin{aligned}
l_{i,i} &= \left(a_{i,i} - \sum_{j=1,(i,j)\in E}^{i-1} l_{i,j}^2 \right)^{\frac{1}{2}} && (3.50)\\
\text{for k} &= i+1, ..., n : && (3.51)\\
&\text{if} \quad (k,i) \in E : && (3.52)\\
l_{k,i} &= \left(a_{k,i} - \sum_{j=1,(k,j)\in E,(i,j)\in E}^{k-1} l_{k,j} l_{i,j} \right) / l_{i,i} && (3.53)\\
&\text{else} \quad : && (3.54)\\
a_{k,k} &= a_{i,i} - \sum_{j=1,(k,j)\in E,(i,j)\in E}^{k-1} l_{k,j} l_{i,j} && (3.55)
\end{aligned}
$$

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.

3.6.5 Effiziente Implementation der modifizierten Cholesky-Zerlegungen

Gerade für diese Aufgabenstellung ist es von großem Interesse, wie der Algorithmus in Code umgesetzt wird. Offensichtlich haben beide Zerlegungen einen Rechenaufwand von $\mathcal{O}(n^2)$. Auch wenn wir nur über das Muster E

iterieren, werden viele Werte der Matrix **A** überprüft. Dies kostet ebenfalls Rechenzeit.
Um dies zu optimieren wird im Fall der Matrix $\mathbf{A}_{2D}$ eine weitere Matrix $\mathbf{B} \in \mathbb{R}^{n\times 5}$ eingeführt. Wie bereits mehrfach erwähnt, enthält die 2D Poisson Matrix maximal 5 Werte ungleich Null pro Zeile. Somit können in der $i-ten$ Zeile maximal 5 Indizes j auftauchen, für die gilt $a_{i,j} \neq 0$. In den Zeilen, in denen die Matrix weniger als fünf Werte ungleich Null hat, wird die $i-te$ Zeile von **B** mit -1 aufgefüllt.

Beispiel:

Um eine Vorstellung von **B** zu bekommen, wählen wir $\mathbf{A}_{2D} \in \mathbb{R}^{9\times 9}$, also $m = 4, N = 3$. Dann folgt:

$$B = \begin{pmatrix} 1 & 2 & (1+N) & -1 & -1 \\ 1 & 2 & 3 & (2+N) & -1 \\ & & \vdots & & \\ (5-N) & 4 & 5 & 6 & (5+N) \\ & & \vdots & & \\ (8-N) & 7 & 8 & 9 & -1 \\ (9-N) & 8 & 9 & -1 & -1 \end{pmatrix} \tag{3.56}$$

Mit der Matrix **B** als Ersatz für das Muster E ergibt sich folgender C++-Code für die modifizierten unvollständige Cholesky-Zerlegung:

3.6.6 C++-Methode der MIC

```
1  void Algorithms::modifiedIncompleteLU(Matrix& A,WriteableMatrix& L,WriteableMatrix& U) {
2      int m,u;
3      double sum, drop;
4
5      for(int i=0;i<dim;i++) {
6          drop=0;
7          for(int k=0;k<5;k++) {
8              m=A.HashMatrix[i][k];
9              if(m!=-1 && m<i) {
10                 sum=0;
11                 for(int j=0;j<5;j++) {
12                     u=A.HashMatrix[i][j];
13                     if(u!=-1 && u<k) {
14                         sum+=L.Get(i,u)*U.Get(u,m);
15                     }
16                 }
17                 L.Set(i,m,(A.Get(i,m)-sum)/U.Get(m,m));
18                 drop+=sum;
19             } else if(m!=-1 && m>=i) {
20                 m=A.HashMatrix[i][k];
21                 if(m!=-1 && m>=i) {
22                     sum=0;
23                     for(int j=0;j<5;j++) {
```

```
24                          u=A.HashMatrix[i][j];
25                          if(u!=-1 && u<i) {
26                              sum+=L.Get(i,u)*U.Get(u,m);
27                          }
28                      }
29                      U.Set(i,m,(A.Get(i,m)-sum));
30                      drop+=sum;
31                  }
32              }
33          }
34          U.Set(i,i,(U.Get(i,i)-drop));
35      }
36  }
```

Analog würde diese Vorschrift für die unvollständige Cholesky-Zerlegung folgen, was wir hier nicht mehr explizit anführen wollen. Betrachtet man in Kapitel 5 die Laufzeiten für den PCG-Algorithmus mit diesen Zerlegungen und der angeführten Implementation, sieht man wie schnell und effizient die Berechnung von statten geht.

4 Mehrgitterverfahren

In diesem Abschnitt sollen nun die Mehrgittermethoden genauer betrachtet werden. Zunächst aber nochmals die Grundlagen:

4.1 Grundlagen

1. Glättungseigenschaft der Jacobi-Relaxation

Das Jacobi-Relaxationsverfahren löscht kurzwellig Fehler in den ersten Iterationsschritten aus. Langwellige Fehler werden nur sehr langsam beseitigt.

2. Residuumsgleichung

Die für diesen Algorithmus wichtige Residuumsgleichung lautet:

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

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

Beweis:

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

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

Der Fehler

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

wobei u^* die exakte Lösung darstellt. Betrachten wir jetzt nochmals Gleichung 4.1, dann stellen wir fest, dass gilt:

$$\mathbf{A}e^k = r^k = b - \mathbf{A}u^k = 0 \text{ für } e^k = 0. \tag{4.4}$$

Somit ist

$$\mathbf{A}e^k = r^k \Longleftrightarrow \mathbf{A}u^k = b \text{ falls } e^k = 0. \tag{4.5}$$

■

4.2 Prolongation

Bevor wir nun auf die Mehrgittermethoden explizit eingehen, müssen wir uns Gedanken darüber machen, wie wir von einem Gitter auf das andere kommen. Angenommen wir befinden uns auf dem groben Gitter Ω_{2h}, so ist das Ziel auf ein feineres Gitter Ω_h mit wenig Rechenaufwand zu kommen und die Werte aus Ω_{2h} sollten auf Ω_h gut genähert abgebildet werden. Für die Prolongation wählen wir hierfür eine lineare Interpolation.

4.2.1 Interpolationsmatrix

Sei $I_{2h}^h : \Omega_{2h} \longrightarrow \Omega_h$ eine Abbildung mit $I_{2h}^h(u_{2h}) = \mathbf{I}u_{2h} = u_h$, wobei $\mathbf{I} \in \mathbb{R}^{(2\tilde{N}-1)^2 \times \tilde{N}^2}$ und $\tilde{N}$ die Anzahl der Gitterpunkte auf dem groben Gitter. Dabei überführt die Matrix $\mathbf{I}$ Vektoren von Ω_{2h} auf Ω_h. Sie ist *nicht* symmetrisch und kann verschiedene Gestalten haben.

$$\mathbf{I} = \frac{1}{4}\begin{pmatrix} I_1 & & & & \\ I_2 & & & & \\ & I_1 & & & \\ & I_2 & & & \\ & & \ddots & & \\ & & & I_1 & \\ & & & I_2 & \end{pmatrix}, \tag{4.6}$$

für $I_1, I_2 \in \mathbb{R}^{(2\tilde{N}-1)\times\tilde{N}}$ gilt folgende Darstellung:

$$\mathbf{I}_1 = \begin{pmatrix} 4 & & & & & \\ 2 & 2 & & & & \\ & 4 & & & & \\ & 2 & 2 & & & \\ & & \ddots & & & \\ & & & 4 & & \\ & & & 2 & 2 & \\ & & & & 4 & \end{pmatrix}, \qquad \mathbf{I}_2 = \begin{pmatrix} 2 & \dots & 2 & & & & \\ 4 & \dots & 4 & & & & \\ & 2 & \dots & 2 & & & \\ & 4 & \dots & 4 & & & \\ & & \ddots & & & & \\ & & & 2 & \dots & 2 & \\ & & & 4 & \dots & 4 & \\ & & & & 2 & \dots & 2 \end{pmatrix}. \tag{4.7}$$

Wir haben hier die Full-Weighted-Matrix der Interpolation verwendet, da sie die höchste Genauigkeit beim Übergang von Ω_{2h} auf Ω_h besitzt. Sie berücksichtigt bei der Interpolation nicht nur Gitterpunkte, die in Ω_{2h}, sowie in Ω_h, existieren, sondern auch den jeweiligen Nachbarn. Es gibt andere Möglichkeiten der Interpolation, z.B. den Half-Weighting-Operator, auf die wir in dieser Arbeit nicht explizit eingehen wollen. Zur Veranschaulichung des Full-Weighted-Operators dient Abbildung 4.2.1. Es geht also in jedes $u_h^{i,j}$ auch der gewichtete Wert aller Nachbarpunkte von $u_{2h}^{i,j}$ des feinen Gitters mit ein.

Beispiel:

$$\mathbf{I}u_{2h}^{i,j} = \frac{1}{4} \begin{pmatrix} I_1 & & & & \\ I_2 & & & & \\ & I_1 & & & \\ & I_2 & & & \\ & & \ddots & & \\ & & & I_1 & \\ & & & I_2 & \end{pmatrix} \begin{pmatrix} u_{2h}^{(1)} \\ \vdots \\ u_{2h}^{(m)} \end{pmatrix} = \begin{pmatrix} u_h^{(1)} \\ u_h^{(2)} \\ \vdots \\ \\ \vdots \\ u_h^{(n-1)} \\ u_h^{(n)} \end{pmatrix} = u_h^{i,j}. \tag{4.8}$$

Bemerkung:

Für den Fall von eindimensionalen Gittern hätte **I** folgende Darstellung:

$$\mathbf{I} = \frac{1}{2}\begin{pmatrix} 1 & & & & & \\ 2 & & & & & \\ 1 & 1 & & & & \\ & 2 & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & 1 & \\ & & & & 2 & \\ & & & & 1 & \end{pmatrix}. \tag{4.9}$$

Hier ist $\mathbf{I} \in \mathbb{R}^{N \times \tilde{N}}$.

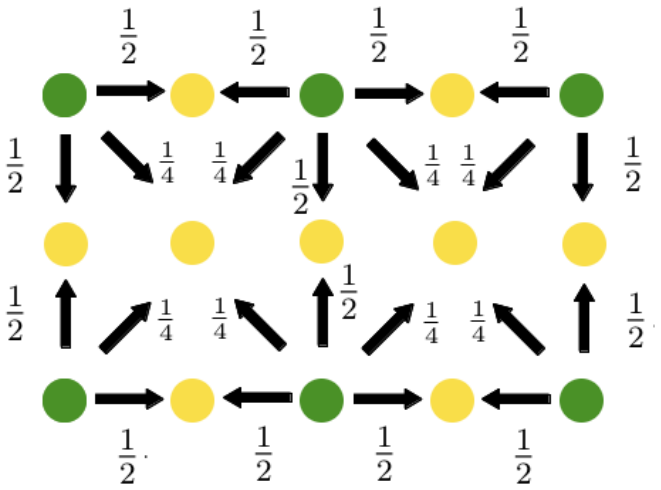

Abbildung 4.1: Bei der Interpolation bedienen sich die Punkte vom feinen Gitter, den gewichteten Punkten des groben Gitters.

Für das Umsetzen in Programmcode ist es natürlich ungünstig eine komplette Matrix-Vektor-Multiplikation zu implementieren, zumal eine Matrix dieser Größe enorm viel Speicherplatz erfordert. Aus diesem Grund lässt sich die Interpolation auch in Komponentenschreibweise fassen:

$$u_h^{2i-1,2j-1} = u_{2h}^{i,j} \qquad i,j = 1,...,\tilde{N}, \tag{4.10}$$

$$u_h^{2i-1,2j} = \frac{1}{2}(u_{2h}^{i,j} + u_{2h}^{i,j+1}) \qquad i = 1,...,\tilde{N}; j = 1,...,\tilde{N}-1, \tag{4.11}$$

$$u_h^{2i,2j-1} = \frac{1}{2}(u_{2h}^{i,j} + u_{2h}^{i+1,j}) \qquad i = 1,...,\tilde{N}-1; j = 1,...,\tilde{N}, \tag{4.12}$$

$$u_h^{2i,2j} = \frac{1}{4}(u_{2h}^{i,j} + u_{2h}^{i,j+1} + u_{2h}^{i+1,j} + u_{2h}^{i+1,j+1}) \qquad i,j = 1,...,\tilde{N}-1. \tag{4.13}$$

Diese Vorschrift lässt sich nun relativ komfortabel und effizient programmieren. Somit ist nun der Übergang vom groben zum feinen Gitter bekannt. Nun wollen wir noch die Gegenrichtung betrachten.

4.3 Restriktion

Sei $R_h^{2h} : \Omega_h \longrightarrow \Omega_{2h}$ mit $R_h^{2h}(u_h) = \mathbf{R}u_h = u_{2h}$ und $\mathbf{R} \in \mathbb{R}^{\tilde{N}^2 \times (2\tilde{N}-1)^2}$. Diese Abbildungsvorschrift nennt man Restriktion. Auch hier gibt es unterschiedliche Methoden, wobei in diesem Abschnitt das Gegenstück zur obigen Interpolation - der Full-Weighting-Operator für die Restriktion - verwendet wird. Auch dieser stellt den exaktesten Übergang zwischen beiden Gittern dar und hat einen speziellen Bezug zur Matrix **I**

$$\mathbf{R} := \frac{1}{4}\mathbf{I}^T \tag{4.14}$$

4.3.1 Restriktionsmatrix

Dadurch ist die Matrixdarstellung gegeben durch:

$$R = \frac{1}{16}\begin{pmatrix} I_1^T & I_2^T & & & \\ & I_1^T & I_2^T & & \\ & & & \ddots & \\ & & & I_1^T & I_2^T \end{pmatrix}, \tag{4.15}$$

wobei I_1^T, I_2^T die transponierten Matrizen von I_1, I_2 darstellen.

Beispiel:

$$\mathbf{R}u_h^{i,j} = \frac{1}{16}\begin{pmatrix} I_1^T & I_2^T & & & \\ & I_1^T & I_2^T & & \\ & & \ddots & & \\ & & & I_1^T & I_2^T \end{pmatrix}\begin{pmatrix} u_h^{(1)} \\ u_h^{(2)} \\ \vdots \\ u_h^{(n)} \end{pmatrix} = \begin{pmatrix} u_{2h}^{(1)} \\ \vdots \\ u_{2h}^{(m)} \end{pmatrix} = u_{2h}^{i,j}. \quad (4.16)$$

Und für die Umsetzung in Code benutzen wir die Komponentenschreibweise:

Für ein u_{2h} auf Ω_{2h} gilt:

$$u_{2h}^{i,j} = \frac{1}{16}(4u_h^{i,j} + 2(u_h^{i+1,j} + u_h^{i-1,j} + u_h^{i,j+1} + u_h^{i,j-1}) + u_h^{i-1,j-1} + u_h^{i-1,j+1} + u_h^{i+1,j+1} + u_h^{i+1,j} \quad (4.17)$$

Das Vorgehen wird von Abbildung 4.3.1 illustriert.

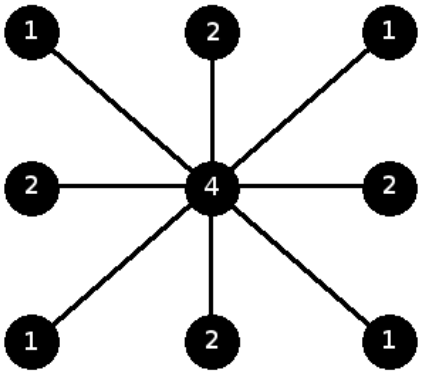

Abbildung 4.2: Ausgehend von einem Punkt innerhalb des Gitters, ist hier die Gewichtung der Werte veranschaulicht. Jeder Wert wird zusätzlich mit einem Faktor $\frac{1}{16}$ multipliziert

Bemerkung:

Auch hier wollen wir noch die Restriktionsmatrix für den eindimensionalen Fall angeben:

$$\mathbf{R} = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & \\ & 1 & 2 & 1 & & & \\ & & & \ddots & & & \\ & & & & 1 & 2 & 1 \end{pmatrix} \tag{4.18}$$

4.4 Transformation der Matrix

Zum Abschluss sollte noch die Matrix $\mathbf{A}$ vom feinen Gitter auf das grobe Gitter transformiert werden. Befänden wir uns nicht auf dem Einheitsquadrat oder hätte $\mathbf{A}$ eine nicht so regelmäßige Struktur wie beispielsweise $\mathbf{A}_{2D}$, dann gilt folgende Transformationsvorschrift:

$$\mathbf{A}_{2h} = \mathbf{R}\mathbf{A}_h\mathbf{I}, \tag{4.19}$$

mit $\mathbf{A}_{2h}, \mathbf{A}_h, \mathbf{R}, \mathbf{I}$ wie oben.
Da wir uns jedoch auf dem Einheitsquadrat befinden und $\mathbf{A}_{2D}$ eine günstige Struktur hat, wollen wir dieses Kapitel nicht weiter vertiefen. Lediglich soll angegeben werden, wie $\mathbf{A}_{2h}$ in unserem Fall nach der Transformation aussieht.

Beispiel:

Sei $N = 7$ die Anzahl der Gitterpunkte in beide Richtungen des feinen Gitters und $\tilde{N} = 3$ die Anzahl der Gitterpunkte in beide Richtungen des groben Gitters. Außerdem seien $\mathbf{A}_{2h} \in \mathbb{R}^{9\times 9}$, $\mathbf{A}_h \in \mathbb{R}^{49\times 49}$, $\mathbf{I} \in \mathbb{R}^{49\times 9}$ und $\mathbf{R} \in \mathbb{R}^{9\times 49}$. So folgt:

$$\mathbf{R}\mathbf{A}_{2D_h}\mathbf{I} = \mathbf{R}\begin{pmatrix} A_1 & -Id & & \\ -Id & A_2 & \ddots & \\ & \ddots & \ddots & \\ & & A_6 & -Id \\ & & -Id & A_7 \end{pmatrix}\mathbf{I} = \begin{pmatrix} A_1 & -Id & \\ -Id & A_2 & -Id \\ & -Id & A_3 \end{pmatrix} = \mathbf{A}_{2D_{2h}} \tag{4.20}$$

Die Matrizen $\mathbf{A}_{2D_{ih}}$ sind also für alle i stets bekannt.

4.5 Einführung in die Mehrgittermethoden

Wie inAbschnitt 2.2 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 des Jacobi-Relaxationsverfahren sind auf diesem Gitter die kurzwelligen 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, k Iterationsschritte aus. So erhalten wir eine Näherung des Fehlers e^k.
Stellt man Gleichung 4.3 um und berechnet $e^k + u^k$, so erhält man eine neue Approximation der exakten Lösung.
Kombiniert man dieses Vorgehen nun mit dem Wechsel zwischen zwei Gittern der Gitterweite h und $2h$ so erhält man das Zweigitterverfahren.

4.6 Das Zweigitterverfahren

Wir wollen das erste Verfahren der Mehrgittermethoden kennen lernen. Der Zweigitter-Algorithmus bildet die Basis für die Mehrgitterverfahren. Die Idee dahinter ist, zunächst die kurzwelligen Fehler durch a priori Fehlerglättung zu eliminieren und auf ein gröberes Gitter zu wechseln. Dort soll dann die Residuumsgleichung gelöst und auf das feinere Gitter zurück gekehrt werden. Hier findet eine a posteriori Fehlerglättung statt. Wiederhole dieses Vorgehen bis zur Konvergenz:

while	$u^k \neq u^*$	
	pre-smooth	JacobiRelaxationMethod
	calculate residual	$r^k = b - Au^k$
	restrict	$r_{2h}^k = Rr_h^k$
		$\mathbf{A}^{2h} = R\mathbf{A}^h P$
	set error	$e_{2h}^0 = 0$
	solve direct	$\mathbf{A}^{2h} e_{2h} = r_{2h}^k$
	prolongate	$e_h^k = Pe_{2h}^k$
	add error	$u_h^k = u_h^{k-1} + e_h^k$
	smooth (optional)	JacobiRelaxationMethod
end		

Es bleiben zwei Fragen nun unbeantwortet:

1. Wie soll die Residuumsgleichung auf dem gröberen Gitter gelöst werden?
2. Wie steht es mit der Konvergenz dieses Verfahrens? Besitzt es die nötige Rechengeschwindigkeit?

Die Antwort auf die Frage nach der Konvergenz werden wir in dieser Arbeit nicht behandeln. Für ein weiteres Studium wird [Saad] empfohlen.

Der klare Nachteil dieser Methode liegt natürlich im direkten Lösen der Residuumsgleichung. Wählen wir ein sehr feines Gebiet Ω_h mit $m = 256$, 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 großen Rechenaufwand, der in dieser Form nicht erwünscht ist.

4.7 Mehrgitter-Algorithmen

Da also das Lösen der Residuumsgleichung auf dem groben Gitter einen direkten Löser erfordert, der zusätzlichen Rechenaufwand bedeutet, ist der

Zweigitter-Algorithmus nicht die Variante, die in der Praxis verwendet wird.

Eine bessere Methode ist, das Gitter immer gröber zu machen, bis das System direkt lösbar ist, um dann wieder auf das feinste Gitter zurück zu kehren. Wir erweitern also das Zweigitterverfahren um Rekursion. Denn ruft sich die Funktion in jedem Iterationsschritt selbst auf und löst direkt, sobald sie auf dem gröbsten Gitter befindet, erhalten wir folgende rekursive Funktion:

V-cycle (u, b)

if *(coarsest grid)*	return $u_{finestgrid} = \mathbf{A}^{-1}b$
else	
pre-smooth	JacobiRelaxationMethod
calculate residual	$r^k = b - Au^k$
restrict	$r_{2h}^k = Rr_h^k$
	$\mathbf{A}^{2h} = R\mathbf{A}^h P$
recursion	$e_{2h}^k = \textbf{V-cycle}(0, r_{2h}^k)$
prolongate	$e_h^k = Pe_{2h}^k$
add error	$u_h^k = u_h^{k-1} + e_h^k$
smooth	JacobiRelaxationMethod
	return u_h

end

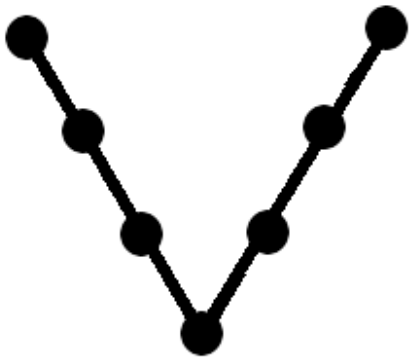

Abbildung 4.3: Text über V-Zyklus

Dieser Algorithmus ist auch als V-Zyklus bekannt. Wie dieser Name zustande kommt illustriert Abbildung 4.7. Nun gibt es eine weitere Variante, den W-Zyklus (illustriert in Abbildung 4.7):

W-cycle (u, b)

if *(coarsest grid)*	return $u_{finestgrid} = \mathbf{A}^{-1}b$
else	
pre-smooth	JacobiRelaxationMethod
calculate residual	$r^k = b - Au^k$
restrict	$r^k_{2h} = Rr^k_h$
	$\mathbf{A}^{2h} = R\mathbf{A}^h P$
recursion	$e^k_{2h} = \mathbf{W\text{-}cycle}(0, r^k_{2h})$
recursion	$e^k_{2h} = \mathbf{W\text{-}cycle}(0, r^k_{2h})$
prolongate	$e^k_h = Pe^k_{2h}$
add error	$u^k_h = u^{k-1}_h + e^k_h$
smooth	JacobiRelaxationMethod
	return u_h

end

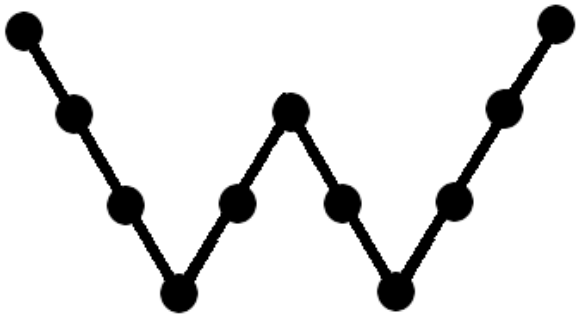

Abbildung 4.4: Text über W-Zyklus

5 Implementierung und Beispiele

In diesem letzten Kapitel sollen noch einmal praktische Code-Beispiele und erhaltene Berechnungswerte anhand eines Beispiels erfolgen. Dafür wollen wir folgende Gleichung betrachten:

5.1 Beispiel einer Poisson Gleichung

Seien $f : \Omega \to \mathbb{R}$ und $g : \partial\Omega \to \mathbb{R}$ 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

$$-\Delta u(x,y) = f(x,y) = -4 \text{ in } \Omega \tag{5.1}$$
$$u(x,y) = g(x,y) = x^2 + y^2 \text{ in } \partial\Omega \tag{5.2}$$

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.
Die gewünschte Lösung sollte also folgenden Graphen ergeben:

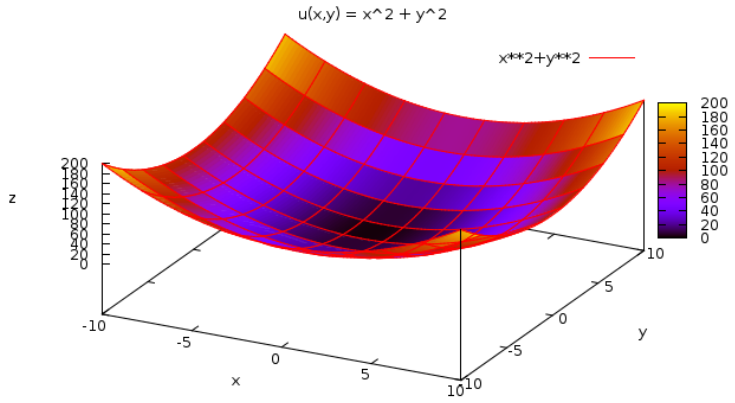

Abbildung 5.1: Die analytische Lösung für diese Poisson-Gleichung war gegeben durch $u(x, y) = x^2 + y^2$.

Zu beachten ist, dass wir $\Omega_h \in [0, 1]$ gewählt haben. Es wird also in den numerischen Lösungen stets der Graph ausgegeben, welcher sich im Einheitsquadrat befindet.

5.2 Zur Implementierung in C++

Das gesamte Programm wurde objektorientiert geschrieben, darum ist von Methoden und Klassen, nicht von Funktionen die Rede. In den folgenden Beispielen wollen wir zunächst die Lösung der Poisson Gleichung für verschiedene Verfahren betrachten. Dafür werden die jeweiligen Methoden der Klassen ebenfalls dargestellt. Außerdem wollen wir nicht nur die Iterationsschritte genauer betrachten, sondern auch die Rechenzeit.
An manchen Stellen im Code kommt die Vermutung auf, dass es sich um Pseudocode handeln könnte. Dies ist natürlich nicht der Fall. Es wurden lediglich bestimmte Operatoren wie *,+,-,etc. überladen.

5.3 Abbruchkriterien

Um zu verstehen, warum wir Abbruchkriterien benötigen, soll hier ein kurzes Beispiel folgen:
Das CG-Verfahren konvergiert bekanntlich nach maximal n Schritten. Wir

brauchen also kein Abbruchkriterium, damit wir die optimale Lösung finden. Angenommen die Dimension der Matrix ist $n = 10^6$. Dann werden trotz der schnellen Konvergenz eine große Anzahl an Iterationen benötigt. Im schlimmsten Fall eben 10^6. Um dies zu vermeiden, lässt man den Algorithmus abbrechen, sobald eine gewisse Toleranzgrenze erreicht ist. In der Praxis schätzt man das $k - te$ Residuum in der A-Norm oder der 2-Norm gegen eine Grenze ab. In diesem Beispiel wählen wir folgenden Ansatz:

$$\|u^k - u^*\|_2 \leq 10^{-3} \cdot \|u^0 - u^*\|_2. \tag{5.3}$$

Im Allgemeinen ist natürlich die Lösung der partiellen Differentialgleichung nicht bekannt. Hier existiert allerdings die analytische Lösung und wir können das Abbruchkriterium so wählen.

5.4 Lösung der Poisson Gleichung (Jacobi-Verfahren)

Wie wir bereits in Unterabschnitt 3.2.4 gesehen haben, konvergiert das Jacobi-Verfahren nur langsam. Es überrascht darum nicht, dass einige Iterationsschritte nötig sind, um das Gleichungssystem zu lösen. Trotz einer effizienten Programmierung benötigt die Methode viel Rechenzeit. Dies illustriert Tabelle 5.4.1.

5.4.1 Jacobi C++-Methode

```
1  void Algorithms::JacobiMethod(Matrix& A,vector<double>& x,const vector<double>& b) {
2      vector<double> r(dim,1),solved(dim);
3
4      for(int i=1,k=0;i<=N;i++) {
5          for(int j=1;j<=N;j++,k++) {
6              solved[k]=g(j*h,i*h);
7          }
8      }
9
10     double TOL=pow(10,-3)*((x-solved)|(x-solved));
11     while(TOL<=(r|r)) {
12         r=b-A*x;
13         x+=1.0/4.0*r;
14     }
15 }
```

N	40	80	160	320
Schritte	2092	8345	33332	133227
Rechenzeit in s	0.45	7.28	120.31	2034.71

Tabelle 5.1: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

5.5 Lösung der Poisson Gleichung (Jacobi-Relaxations-Verfahren)

Zunächst soll der C++-Code angegeben werden, da er für beide Parameter identisch ist. Lediglich der Parameter ω ändert sich.

5.5.1 Jacobi-Relaxation C++-Methode

```
1  void Algorithms::JacobiRelaxationMethod(Matrix& A,vector<double>& x,const vector<double>& b) {
2      vector<double> r(dim,1),solved(dim);
3      double omega=4.0/5.0;
4
5      for(int i=1,k=0;i<=N;i++) {
6          for(int j=1;j<=N;j++,k++) {
7              solved[k]=g(j*h,i*h);
8          }
9      }
10
11     double TOL=pow(10,-3)*((x-solved)|(x-solved));
12     while(TOL<=(r|r)) {
13         r=b-A*x;
14         x+=(omega*1.0/4.0)*r;
15     }
16 }
```

5.5.2 Parameter $\omega = \frac{1}{2}$

In Unterabschnitt 3.3.3 haben wir gesehen, das der Spektralradius des Jacobi-Relaxationsverfahrens näher an eins liegt, als der des Jacobi-Verfahrens. Aus diesem Grunde benötigt das relaxierte Verfahren auch wesentlich mehr Iterationsschritte. Nun ist klar, dass es als iteratives Verfahren gänzlich ungeeignet ist und lediglich als Glätter für die Mehrgittermethoden dient.

N	40	80	160	320
Schritte	4186	16693	66667	266456
Rechenzeit in s	0.88	14.43	237.10	3068.42

Tabelle 5.2: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

Das Erstaunliche an diesen Messwerten ist nun, dass man für den Parameter $\omega = \frac{1}{2}$ stets doppelt so viele Iterationsschritte benötigt, als beim Jacobi-Verfahren.

5.5.3 Parameter $\omega = \frac{4}{5}$

Folgt man nun der Logik, dass sich bei einem Faktor $\omega = \frac{1}{2}$ die Iterationswerte verzweifachen, so müssten die Iterationsschritte für den Parameter $\omega = \frac{4}{5}$ ca. 1.25 mal so viele sein, als beim Jacobi-Verfahren.

N	40	80	160	320
Schritte	2615	10432	41666	166534
Rechenzeit in s	0.55	9.00	147.91	3794.98

Tabelle 5.3: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

Teilt man nun die Werte der Iterationsschritte aus Tabelle 5.5.3 durch die von Tabelle 5.4.1, erhält man tatsächlich einen Faktor von ≈ 1.25. Diese Tatsache liegt natürlich daran, dass das aufaddierte Residuum bei jedem Iterationsschritt, um den Faktor ω verkleinert wird.

5.6 CG-Verfahren angewandt auf das Beispiel

Da das CG-Verfahren eines der effizienteren Iterationsverfahren ist, sollte natürlich die Messwerte dementsprechend gut sein. Man sieht aber im Folgenden die schlechte Kondition von $\mathbf{A}_{2D}$, je feiner das Gitter wird. Pro

Verdopplung der Gitterweite, verdoppeln sich auch die Iterationsschritte. Die Rechenzeit für ein sehr feines Gitter ist ebenfalls nicht akzeptabel.

N	40	80	160	320
Schritte	65	130	262	525
Rechenzeit in s	0.02	0.16	1.35	11.26

Tabelle 5.4: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

Der C++-Code für das CG-Verfahren, wie auch im nächsten Abschnitt das PCG-Verfahren ist verhältnismäßig lang und würde den Rahmen dieses Kapitels sprengen. Die Implementierung folgt aber eins zu eins dem Algorithmus aus Unterabschnitt 3.5.10 und Unterabschnitt 3.6.2.

5.7 PCG-Verfahren angewandt auf das Beispiel

In diesem Abschnitt müsste man nun eine deutliche Verbesserung für die Anzahl der Iterationsschritte und der Rechenzeit sehen.

5.7.1 Unvollständige Cholesky-Zerlegung

Auch wenn man eine klare Verbesserung zum Standardverfahren der konjugierten Gradienten sieht, sind die Ergebnisse immer noch nicht in einem akzeptablen Rahmen. Speziell für das feinste Gitter ist die Rechenzeit nur unwesentlich besser als in Tabelle 5.6.

N	40	80	160	320
Schritte	20	40	79	158
Rechenzeit in s	0.01	0.14	1.1	8.78

Tabelle 5.5: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

5.7.2 Modifizierte unvollständige Cholesky-Zerlegung

Für den Fall der modifizierten unvollständigen Cholesky-Zerlegung erwartet man nun eine deutlichen Effekt auf Anzahl der Iterationsschritte und Rechenzeit.

N	40	80	160	320
Schritte	7	10	13	19
Rechenzeit in s	0.00	0.04	0.21	1.23

Tabelle 5.6: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

Wie Tabelle 5.7.2 nun eindrucksvoll zeigt, liegt die Rechenzeit für das feinste Gitter gerade mal mehr als einer Sekunde. Dies ist natürlich eine enorme Verbesserung zum Anfangsproblem. Mit 19 Iterationsschritten liegt man hierbei deutlich unter der Anzahl für das gröbste Gitter beim CG-Verfahren.

5.8 Das Mehrgitterverfahren angewandt auf das Beispiel

5.8.1 V-Zyklus

N	32	64	128	256
Schritte	13	12	12	11
Rechenzeit in s	0.02	0.07	0.31	1.20

Tabelle 5.7: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

5.8.2 W-Zyklus

N	32	64	128	256
Schritte	11	11	10	9
Rechenzeit in s	0.03	0.11	0.45	2.03

Tabelle 5.8: Je größer N wird, desto mehr Iterationsschritte und Rechenaufwand ist zum Lösen der Gleichung nötig.

Es zeigt sich also, dass sowohl iterative Verfahren, als auch die Mehrgittermethoden, in der numerischen Mathematik eine tragende Rolle spielen. Die Effizienz dieser Verfahren ist beeindruckend und findet in vielen Gebieten Anwendung. Man darf gespannt sein, was in Zukunft noch für Ansätze zur Lösung linearer Gleichungssysteme gefunden werden. Allerdings ist es nur schwer vorstellbar, noch effizientere Algorithmen als die vorgestellten zu entwickeln.

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