UNIVERSITÄT REGENSBURG

Fakultät für Mathematik

Bachelor of Science Computational Science

Bachelorarbeit

zur Erlangung des akademischen Grades eines Bachelor of Science (B. Sc.)

Iterative Verfahren zur Lösung der diskretisierten Poisson Gleichung

in der angewandten Mathematik.

von

Michael Bauer

Matrikelnummer: 152 8558

Betreuer: Prof. Dr. Harald Garcke

Eingereicht am: 20.03.2014

Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Ich habe die Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und bisher keiner anderen Prüfungsbehörde vorgelegt. Außerdem bestätige ich hiermit, dass die vorgelegten Druckexemplare und die vorgelegte elektronische Version der Arbeit identisch sind, dass ich über wissenschaftlich korrektes Arbeiten und Zitieren aufgeklärt wurde und dass ich von den in § 26/27 Abs. 5 vorgesehenen Rechtsfolgen Kenntnis habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Viele Prozesse in den Naturwissenschaften, wie Biologie, Chemie und Physik, aber auch der Medizin und Wirtschaft lassen sich auf partielle Differentialgleichungen zurückführen. Das Lösen solcher Gleichungen ist allerdings nicht immer möglich, oder aufwendig.
Eine partielle Differentialgleichung, die vor allem in der Physik häufige Verwendung findet, ist die Poisson-Gleichung. Sie stellt eine elliptische partielle Differentialgleichung zweiter Ordnung dar. So genügt beispielsweise das elektrostatische Potential u zu gegebener Ladungsdichte f, oder das Gravitationspotential u zu gegebener Massendichte f dieser Gleichung. Wie man sieht, hängt diese Gleichung mit ganz elementaren Dingen unseres Lebens zusammen.
Methoden aus der numerischen Mathematik ermöglichen das Lösen von partiellen Differentialgleichungen mittels computerbasierten Algorithmen. Es 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, also durch Stabilität und geringem Rechenaufwand gekennzeichnet ist.
Unser Ziel ist es nun, solche Algorithmen herzuleiten und in Programmiercode umzusetzen, damit wir die oben genannte Poisson Gleichung im zweidimensionalen Raum lösen können. Bevor man die Lösung einer partiellen Differentialgleichung berechnen kann, versucht man zunächst die Gleichung auf ein lineares Gleichungssystem $\mathbf{A}u = f$ zurückzuführen. Eine der zentralen Methoden der Numerik sind Finite-Differenzen. Hierbei diskretisiert man das Gebiet, auf dem die partielle Differentialgleichung definiert ist und kann dann die Gleichung auf ein lineares Gleichungssystem der Form $\mathbf{A}u = f$ zurück führen.
Da es eine Reihe von Methoden zur Lösung von linearen Gleichungssystemen gibt, stellt sich natürlich die Frage, welche die effizienteste für die gegebene Problemstellung ist. Eine Möglichkeit, ein lineares Gleichungssystem zu lösen, sind iterative Verfahren. Sie zeichnen sich dadurch aus, dass sie die approximierte Lösung schrittweise nähern. Dabei werden pro Iterationsschritt nur endlich viele Rechenoperationen benötigt. Beispiele, die in dieser

Arbeit diskutiert werden, sind das Jacobi-Verfahren oder das Verfahren der konjugierten Gradienten. Letzteres liefert ein großartiges Werkzeug, um eine Lösung innerhalb weniger Iterationsschritte zu berechnen.
Wie wir sehen werden, ist das Jacobi-Verfahren zwar ein iteratives Verfahren, jedoch als solches ungeeignet. Durch eine Modifikation erhalten wir das Jacobi-Relaxationsverfahren, welches ebenfalls nicht als ein iteratives Verfahren geeignet ist, allerdings die sogenannte Glättungseigenschaft besitzt. Diese Eigenschaft findet ihre Anwendung in den Mehrgittermethoden, welche sich durch schnelle Konvergenz und geringe Rechenzeit auszeichnen. Gerade für die Poisson-Gleichung liefern die Mehrgittermethoden innerhalb weniger Sekunden eine sehr gute Approximation der Lösung.
Um einen Vergleich dieser Verfahren zu erhalten, wollen wir abschließend diese miteinander vergleichen. Dabei betrachten wir, basierend auf einem selbst entwickelten C++-Algorithmus, die Iterationsschritte für die iterativen Verfahren und die Anzahl der Zyklen bei den Mehrgittermethoden. Zusätzlich soll uns die Rechenzeit als Referenz beim Vergleich der Verfahren dienen und uns deutliche Unterschiede erkennen lassen. So werden wir sehen, dass speziell für die Poisson Gleichung die Mehrgittermethoden die stärksten Algorithmen darstellen.

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

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

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

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

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

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

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 den zentralen Differenzenquotienten (zweiter Ordnung) einführen, Mithilfe dessen wir bei der Diskretisierung eine Matrix **A** erhalten werden. Diese wollen wir auf ihre Eigenschaften untersuchen.

2.2.1 Zentraler Differenzenquotient zweiter Ordnung

In diesem Abschnitt folgen wir [ALO1].
Wir betrachten ein $(x, y) \in \Omega$ beliebig. Dann gilt für $u(x, y)$ mit $h > 0$ und der Taylorformel

$$u(x+h,y) = u(x,y) + h\partial_x u(x,y) + \frac{h^2}{2!}\partial_{xx}u(x,y) + \frac{h^3}{3!}\partial_{xxx}u(x,y) + \frac{h^4}{4!}\partial_{xxxx}u(\xi,y), \quad (2.5)$$

$$u(x-h,y) = u(x,y) - h\partial_x u(x,y) + \frac{h^2}{2!}\partial_{xx}u(x,y) - \frac{h^3}{3!}\partial_{xxx}u(x,y) + \frac{h^4}{4!}\partial_{xxxx}u(\zeta,y), \quad (2.6)$$

wobei ξ und ζ jeweils unbekannte Zwischenstellen zwischen x und $x \pm h$ sind. Addieren wir beide Gleichungen und teilen durch h^2, so ergibt dies:

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

Analog können wir für $u(x, y+h)$ und $u(x, y-h)$ vorgehen und erhalten:

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

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

$$-\Delta u(x,y) = \frac{u(x-h,y) - u(x+h,y) + 4u(x,y) - u(x,y-h) - u(x,y+h)}{h^2}. \quad (2.9)$$

2.2.2 Diskretisierung von Ω

Mit einem zweidimensionalen Gitter, der Gitterweite h, wobei $h \in \mathbb{Q}$ mit $h = \frac{1}{m}$ und $m \in \mathbb{N}_{>2}$, wird nun das Gebiet Ω diskretisiert. Die Zahl $N = (m-1)$ gibt an, wie viele innere Gitterpunkte es jeweils in x- bzw. y-Richtung gibt.
Die folgenden Aussagen sind weitestgehend in [DR1] wieder zu finden.

Für $i, j = 1, ..., N$ fassen wir $u(x, y)$ auf als:

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

Ω schreiben wir als Ω_h, so dass gilt:

$$\Omega_h = \{(ih, jh) | 1 \leq i, j \leq N\}. \tag{2.11}$$

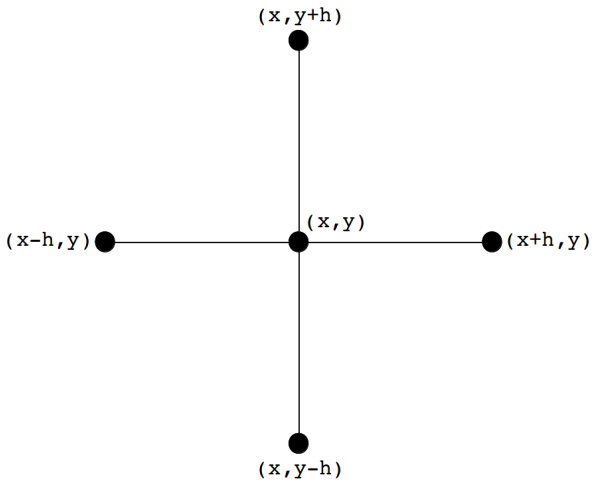

Abbildung 2.1: 5-Punkt-Differenzenstern im Gitter, wobei für alle $1 < i, j < N$ jeder Punkt $(x, y) \in \Omega_h$ genau vier Nachbarn in Ω_h besitzt.

Mit Gleichung 2.9 fassen wir $\Delta u(x, y) = \Delta_h u(x, y)$ für alle $(x, y) \in \Omega_h$ in diskretisierter Form auf als:

$$-\Delta_h u(x, y) = \frac{u(x-h, y) - u(x+h, y) + 4u(x, y) - u(x, y-h) - u(x, y+h)}{h^2}. \tag{2.12}$$

Man stellt $\Delta_h u(x,y)$ auch als 5-Punkt-Differenzenstern (Abbildung 2.1) in der Form

$$[-\Delta_h]_\xi = \frac{1}{h^2} \begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix}_\xi , \xi \in \Omega_h \tag{2.13}$$

dar.

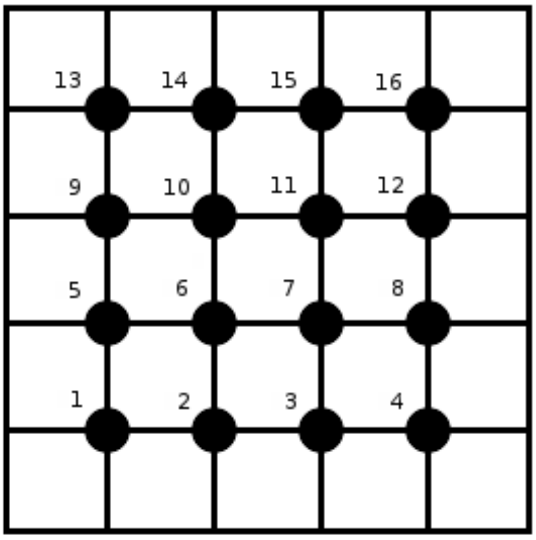

Abbildung 2.2: Nummerierung der Punkte von $\Omega = (0,1)^2$ mit $m = 5$. In x- bzw. y-Richtung gibt es jeweils $N = 4$ Punkte.

Es werden nun alle Gitterpunkte des Gitters fortlaufend von links unten nach rechts oben (Abbildung 2.2) nummeriert (Lexikographische Nummerierung). Stellt man nun Gleichung 2.12 für jeden Punkt auf, so führt dies auf eine $N^2 \times N^2$-Matrix, in der die Koeffizienten der Gleichung abgespeichert werden. Die vierfache Gewichtung der Funktion $u(x,y)$ steht in der Diagonale. Der linke bzw. der rechte Nachbar im Gitter sind auf der unteren bzw. oberen Nebendiagonalen zu finden, so fern der Nachbar in Ω_h liegt. Die oberen bzw. unteren Nachbarn, falls diese in Ω_h liegen, werden jeweils auf der $N - ten$ Nebendiagonalen der Matrix gespeichert.

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

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

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

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

Nun wissen wir, wie $-\Delta u_h$ in Matrixform aussieht. Man nennt eine Matrix bei der viele Einträge gleich Null sind dünn besetzt oder sparse. Die rechte Seite f der partiellen Differentialgleichung kann nun ebenfalls in diskretisierter Form f_h geschrieben werden. Zu jeder Komponente von f_h, die einen Randpunkt als Nachbarn hat, wird dieser dazu addiert. Hat eine Komponente von f_h zwei Nachbarn auf $\partial\Omega_h$, werden beide addiert. Dies führt uns auf folgende rechte Seite, wobei wir diese nun als f auffassen wollen:

$$f = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}, \tag{2.16}$$

wobei gilt

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

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

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

Somit ergibt sich das lineare Gleichungssystem $\mathbf{A}u = f$, wobei $\mathbf{A}$ der diskretisierte Laplace-Operator ist, f die rechte Seite der Gleichung darstellt und u die approximierte Lösung der partiellen Differentialgleichung enthält.
Wir bezeichnen ab jetzt die zweidimensionale Poisson Matrix aus Gleichung 2.14 mit $\mathbf{A}_{2D}$. Außerdem gilt für die Schrittweite stets $h = \frac{1}{m}$, wobei $m \in \mathbb{N}_{m>2}$. Zudem setzen wir $N = m - 1$ und $n = N^2$, so dass wir die Poisson Matrix als eine Matrix $\mathbf{A}_{2D} \in \mathbb{R}^{n \times n}$ auffassen werden.

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

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

$$\mathbf{A}_{2D} v_i = \lambda_i v_i, \tag{2.20}$$

wobei für alle $1 \leq i \leq n$ gilt: $\lambda_1, ..., \lambda_n \in \mathbb{R}$ und $v_i \in \mathbb{R}^n$.

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

Aus [SAAD1] sei für $k, l \in \mathbb{N}$, $(x_k, y_l) \in \Omega_h$ mit $x_k = k \cdot h, y_l = l \cdot h$. Zudem sind $\theta_k, \theta_l \in \mathbb{R}$ mit $\theta_k = k\pi h$ bzw. $\theta_l = l\pi h$. Dann gilt für die Eigenwerte:

$$\lambda_{k,l} = 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) \text{ für } 1 \leq k, l \leq N. \tag{2.21}$$

Zudem gilt für einen Eigenvektor am Punkt (x_k, y_l) gilt:

$$(v_{k,l})_{i,j} = \sin(i\pi x_k)\sin(j\pi y_l) = \sin(i\theta_k)\sin(j\theta_l) \text{ für } 1 \leq i, j, k, l \leq N. \tag{2.22}$$

Bemerkung:

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

$$v_{k.l} = \begin{pmatrix} \sin(\theta_k)\sin(\theta_l) \\ \sin(\theta_k)\sin(2\theta_l) \\ \vdots \\ \sin(\theta_k)\sin(N\theta_l) \\ \sin(2\theta_k)\sin(\theta_l) \\ \sin(2\theta_k)\sin(2\theta_l) \\ \vdots \\ \sin(N\theta_k)\sin((N-1)\theta_l) \\ \sin(N\theta_k)\sin(N\theta_l) \end{pmatrix}. \tag{2.23}$$

Beispiel:

Um sich die Eigenvektoren besser vorstellen zu können, wollen wir zunächst drei Eigenvektoren des eindimensionalen Poisson Problems betrachten. Es wird das Gebiet $\Omega_h = (0, 1)$ diskretisiert und es gilt $u(x)'' = f(x)$ mit Randbedingung $u(x) = g(x)$. Dann erhält man auf einem Gitter $(0, 1)$ mit $h = \frac{1}{m}$ wie oben, die Eigenvektoren des diskretisierten Problems in den Punkten 1,7 und 13:

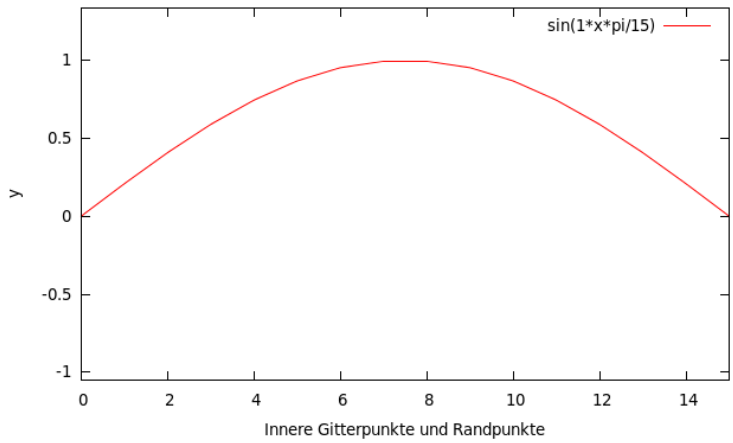

Abbildung 2.3: Dieser Graph stellt den Eigenvektor am ersten Punkt, also $h = \frac{1}{15}$ dar. Die Sinuswellen sind zwar eher kurzwellig, allerdings nicht stark oszillierend.

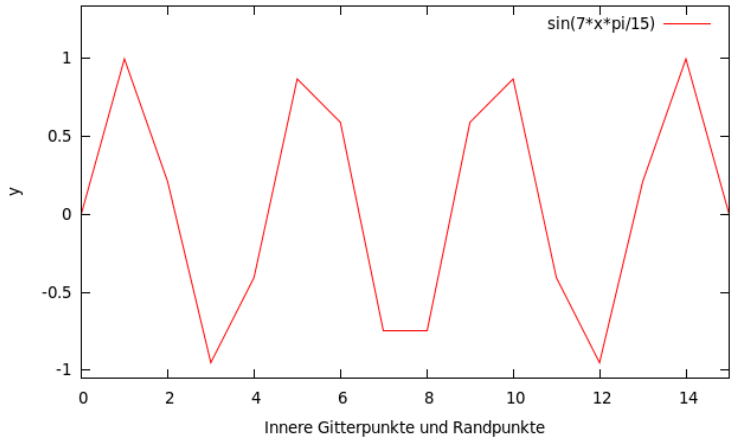

Abbildung 2.4: Bei diesen stark oszillierenden Sinuswellen handelt es sich um den Eigenvektor im Punkt $\frac{7}{15}$.

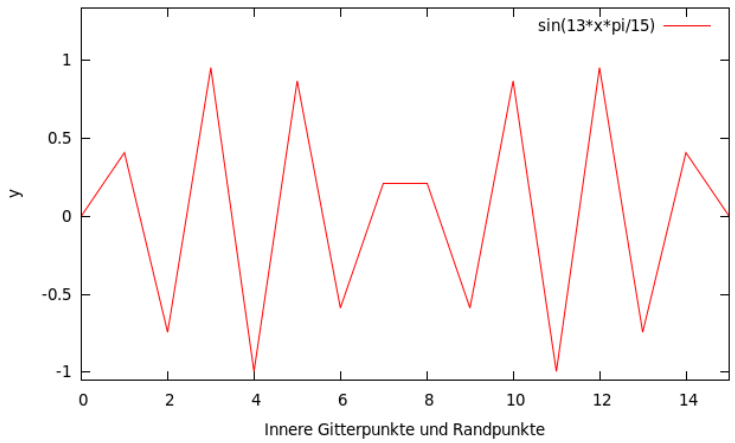

Abbildung 2.5: Die Sinuswellen sind für den Eigenvektor im Punkt $\frac{13}{15}$ kaum mehr erkennbar, wegen ihrer starken Oszillation.

Man sieht deutlich, dass die Oszillation der einzelnen Eigenvektoren unterschiedlich ist. Manche der Eigenvektoren sind langwellig, andere kurzwellig. Wenn wir nun zurück zu unserer Ausgangssituation des zweidimensionalen Poisson Problems gehen und das Gitter aus Abbildung 2.2 als Basis nehmen, erhalten wir für den Punkt 11 des Gitters folgende Darstellung des Eigenvektors:

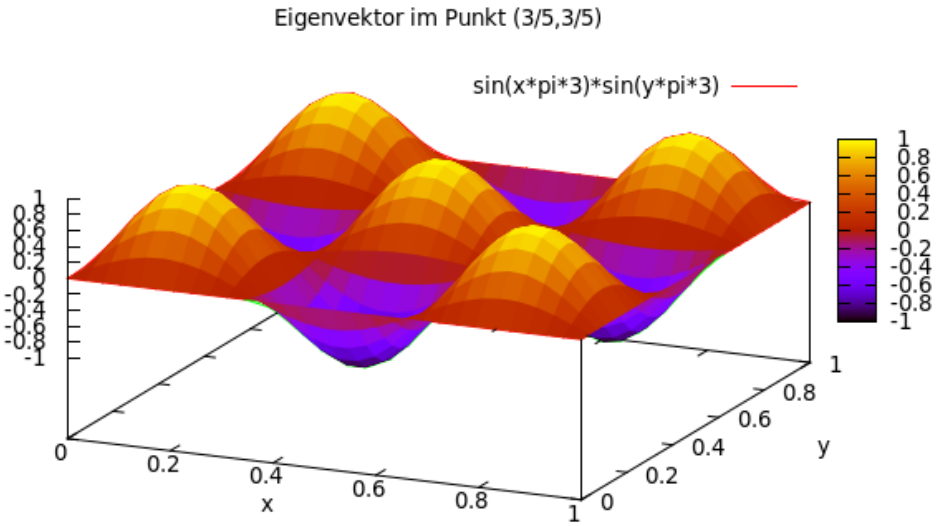

Abbildung 2.6: Es sind die Sinuswellen des Eigenvektors im Gitterpunkt $(\frac{3}{5}, \frac{3}{5})$ zu sehen. Sie sind langwellig und nur schwach oszillierend.

Zur besseren Darstellung wurde für den Plot der Eigenvektor das kontinuierlichen Problem herangezogen. Man konnte hier einen langwelligen Eigenvektor erkennen. Von besonderem Interesse sind in späteren Abschnitten jedoch die kurzwelligen, stark oszillierenden Eigenvektoren.

Beweis zu Unterabschnitt 2.3.1:

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

$$\mathbf{B} = \begin{pmatrix} a & b & & & \\ c & a & b & & \\ & \ddots & \ddots & \ddots & \\ & & c & a & b \\ & & & c & a \end{pmatrix} \tag{2.24}$$

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 $1 \leq i \leq N$, $\lambda_k \in \mathbb{R}, v_k \in \mathbb{R}^N$.

Beweis:

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

$$(\mathbf{B} - \lambda_k \mathbf{Id}) v_k = 0 \tag{2.25}$$

$$\iff \begin{pmatrix} a-\lambda_k & b & & & \\ c & a-\lambda_k & b & & \\ & \ddots & \ddots & \ddots & \\ & & c & a-\lambda_k & b \\ & & & c & a-\lambda_k \end{pmatrix} v_k = 0 \tag{2.26}$$

$$\iff \begin{pmatrix} (a-\lambda_k)v_{k_1} + bv_{k_2} \\ cv_{k_1} + (a-\lambda_k)v_{k_2} + bv_{k_3} \\ \vdots \\ cv_{k_{N-2}} + (a-\lambda_k)v_{k_{N-1}} + bv_{k_N} \\ cv_{k_{N-1}} + (a-\lambda_k)v_{k_N} \end{pmatrix} = 0. \tag{2.27}$$

Wir wollen zunächst die einzelnen Summanden ausrechnen bzw. vereinfachen. Dabei benutzen wir u.a. die Additionstheoreme:

1. Löse $(a-\lambda_k)(v_k)_i$

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

2. Löse $b(v_k)_{i+1}$

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

3. Löse $c(v_k)_{i-1}$

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

Nun rechnen wir jede Zeile unseres Vektors aus und nutzen dabei wieder die Additionstheoreme:

1. $i = 1$

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

2. 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. $i = N$

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

■

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 Eigenwerten $\mu_k = -1$, denn

$$(-\mathbf{Id} - \lambda_k \mathbf{Id})v_k = 0. \tag{2.28}$$

Nun wollen wir diese Erkenntnisse für die Matrix $\mathbf{A}_{2D} \in \mathbb{R}^{n \times n}$ verwenden. Da diese Matrix ebenfalls eine Tridiagonalmatrix ist, folgt für $a = A_j$, wobei $1 \leq j \leq N$ und $b = c = -Id$.
Der gesuchte Eigenvektor in einem Punkt (x_k, y_l) war gegeben durch $(v_{k,l})_{i,j} = \sin(i\theta_k)\sin(j\theta_l)$ für alle $1 \leq i, j, k, l \leq N$. Diese Eigenvektoren können wir auch auffassen als $(v_{k,l})_{i,j} = \sin(j\theta_l)(v_k)_i$ für alle $1 \leq i, j, k, l \leq N$. Wegen $\mathbf{A}_{2D}$ symmetrisch folgt:

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

Für $1 \leq l, k \leq N$ werden wir nun $\mathbf{A}_{2D}v_{k,l}$ explizit ausrechnen und müssen dafür wieder drei Fälle unterscheiden, bei denen wir die Additionstheoreme benutzen:

1. $j = 1$

$$
\begin{aligned}
& \underbrace{A_1 v_k}_{=\lambda_k v_k} \sin(\theta_l) + \underbrace{(-Id v_k)}_{=\mu_k v_k} \underbrace{\sin(2\theta_l)}_{=\sin(\theta_l+\theta_l)=2\cos(\theta_l)\sin(\theta_l)} \\
= \;& \lambda_k v_k \sin(\theta_l) + 2\mu_k v_k \cos(\theta_l)\sin(\theta_l) \\
= \;& (\lambda_k + 2\mu_k \cos(\theta_l)) \underbrace{v_k \sin(\theta_l)}_{=(v_{k,l})_{j=1}} = (\lambda_k + 2\mu_k \cos(\theta_l))(v_{k,l})_{j=1} \\
= \;& (4(1 - \frac{1}{2}\cos\theta_k) - 2\cos(\theta_l))(v_{k,l})_{j=1} \\
= \;& (4 - 2\cos(\theta_k) - 2\cos(\theta_l))(v_{k,l})_{j=1} \\
= \;& (2 - 2\cos(\theta_k) + 2 - 2\cos(\theta_l))(v_{k,l})_{j=1} \\
= \;& \underbrace{(4(\frac{1}{2} - \frac{1}{2}\cos(\theta_k)) + 4(\frac{1}{2} - \frac{1}{2}\cos(\theta_l))}_{\text{mit } \frac{1}{2}(1-\cos(x))=\sin^2(\frac{x}{2})} (v_{k,l})_{j=1} \\
= \;& 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right)(v_{k,l})_{j=1}
\end{aligned}
$$

2. für alle $2 \leq j \leq N-1$

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

3. $j = N$

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

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 die Eigenwerte $\lambda_{k,l} \in \mathbb{R}$ von $\mathbf{A}_{2D}$ mit $1 \leq k, l \leq N$ gilt:

$$\lambda_{k,l} > 0, \tag{2.29}$$

d.h. die Poisson Matrix ist symmetrisch positiv definit.

Beweis:

Da $\sin^2(x) \in (0,1)$ streng monoton steigend ist für $x \in (0, \frac{\pi}{2})$ und $0 < \frac{\pi h}{2} < \frac{\pi}{2}$ gilt:

$$\lambda_{k,l} = 4\left(\sin^2(\frac{\theta_k}{2}) + \sin^2(\frac{\theta_l}{2})\right) > 0. \tag{2.30}$$

■

2.3.3 Definition (Kondition einer symmetrischen Matrix)

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

$$\kappa_2(A) = \frac{\lambda_{max}}{\lambda_{min}} \geq 1. \tag{2.31}$$

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

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

In [DR2] wird gezeigt, dass für die Kondition der Matrix $\mathbf{A}_{2D}$ gilt:

$$\kappa_2(A_{2D}) = \frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})}. \tag{2.32}$$

Beweis:

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

$$\lambda_{min} = \lambda_{1,1} = 4(\sin^2(\frac{\pi h}{2}) + \sin^2(\frac{\pi h}{2})) = 8\sin^2(\frac{\pi h}{2}), \tag{2.33}$$

$$\begin{aligned}
\lambda_{max} &= \lambda_{N,N} = 8\sin^2(\frac{N\pi h}{2}) \overset{\substack{h=\frac{1}{m},\\ N=m-1}}{=} 8\sin^2(\frac{(m-1)\pi}{2m}) \\
&= 8\sin^2(\frac{\pi}{2} - \frac{\pi}{2m}) = 8\sin^2(\frac{\pi}{2} - \frac{\pi h}{2}) \\
&= 8(\sin(\frac{\pi}{2})\cos(\frac{\pi h}{2})) - \sin(\frac{\pi h}{2})\cos(\frac{\pi}{2}))^2 \\
&= 8\cos^2(\frac{\pi h}{2}).
\end{aligned} \tag{2.34}$$

Somit folgt aus Gleichung 2.33 und Gleichung 2.34:

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

■

In [DR3] wird außerdem gezeigt, dass sich $\kappa_2(\mathbf{A}_{2D})$ wie folgt nähern lässt:

$$\frac{\cos^2(\frac{\pi h}{2})}{\sin^2(\frac{\pi h}{2})} = \left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)). \tag{2.35}$$

Natürlich wollen wir die Funktion $u(x, y)$ so gut wie möglich in Ω_h approximieren. Wir 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.

$$\left(\frac{2}{\pi h}\right)^2 (1 + \mathcal{O}(h^2)) = \left(\frac{2}{\pi \frac{1}{m}}\right)^2 (1 + \mathcal{O}(\frac{1}{m^2})) \approx \frac{4m^2}{\pi^2}. \tag{2.36}$$

Man sieht, dass die Kondition der Matrix quadratisch mit m wächst.

3 Iterative Lösungsverfahren für lineare Gleichungssysteme

Gleichungssysteme, die partielle Differentialgleichungen lösen, sind im Allgemeinen sehr groß. Aus diesem Grund sind direkte Verfahren, wie z.B. der Gauß-Algorithmus oder die LR-Zerlegung nicht geeignet. Ihr Rechenaufwand beläuft sich für voll besetzte Matrizen im Allgemeinen auf $\mathcal{O}(n^3)$ und ist zu langsam. Nun ist unser System zwar nicht voll besetzt, jedoch kann es durch die Struktur der Matrix zur Auslöschung von Nullen kommen. Dadurch ist ebenfalls keine große Effizienz gegeben.
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 [DR4]. Der Abschnitt über das Jacobi-Relaxationsverfahren wurde in Teilen von [SAAD2] übernommen. Stellen, an denen andere Literatur verwendet wurde, sind deutlich gekennzeichnet.

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 positiv definit, 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 beliebigem Startvektor $u^0 \in \mathbb{R}^n$ konvergiert genau dann gegen die exakte Lösung $u^* \in \mathbb{R}^n$, wenn gilt:

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

Je kleiner ρ ist, desto schneller konvergiert das System.

Einen Beweis hierzu findet man z.B. in [DR5].

3.1.4 Definition (Residuum und Fehler)

Sei $u^* \in \mathbb{R}^n$ die exakte Lösung des linearen Gleichungssystems $Au = f$. Sei weiterhin $u^k \in \mathbb{R}^n$ 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:

$$\begin{aligned} & e = u^* - u \Leftrightarrow \mathbf{A}e = \mathbf{A}(u^* - u) \\ \Leftrightarrow \quad & \mathbf{A}e = \mathbf{A}u^* - \mathbf{A}u \Leftrightarrow \mathbf{A}e = f - \mathbf{A}u \\ \Leftrightarrow \quad & \mathbf{A}e = r. \end{aligned} \tag{3.7}$$

$\mathbf{A}e = r$ 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 $\mathbf{D}, \mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times n}$, mit $\mathbf{D} = diag(a_{1,1}, ..., a_{n,n})$, $\mathbf{L}$ strikte untere und $\mathbf{U}$ strikte obere Dreiecksmatrix.
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 [ALO2] wird gezeigt, wie man den Rechenaufwand verbessern kann, indem man die Struktur von $\mathbf{A}_{2D}$ optimal ausnutzt.
Mit einem Startvektor $u^0 \in \mathbb{R}^n$ *beliebig* und $k = 1, 2, ...$, berechne für $i = 1, ..., n$:

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

In jedem Schritt zur Berechnung von u^{k+1} muss im Algorithmus 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 von $\mathbf{T}_J$ 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 von $\mathbf{T}_J$ bzgl. $\mathbf{A}_{2D}$)

Das Jacobi-Verfahren konvergiert für die diskretisierte 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 $\omega = 1$.

■

Wir können für das Jacobi-Verfahren zwar die Konvergenz nicht verbessern, allerdings ist es möglich den Rechenaufwand zu verbessern. Dies gelingt uns, indem wir die Dünnbesetztheit von $\mathbf{A}_{2D}$ gezielt ausnutzen. 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 $\mathbf{A}_{2D}$)

Dieser Abschnitt wurde sinngemäß aus [ALO2] übernommen. Der Algorithmus wurde auf unser Problem hingehend optimiert. Berechne für $k = 1, 2, ...$ mit Startvektor $u^0 \in \mathbb{R}^n$ *beliebig*:
Für $i = 1, ..., N$ und für $j = 1, ..., N$:

$$u_{i,j}^{k+1} = \frac{h^{-2}}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k) - \frac{1}{4}f_{i,j} \tag{3.15}$$

Bei diesem Algorithmus gehen wir über allen inneren Punkte des Gitters in x- und y-Richtung. Sobald ein Zeilenindex i (bezeichnet die y-Komponente) Null wird, ignorieren wir diesen Wert. Das Gleiche gilt für die Spalteninidzes j, die die x-Komponente repräsentieren.
Der Rechenaufwand pro Iterationsschritt beträgt lediglich $\mathcal{O}(N \cdot N) = \mathcal{O}(n)$ Schritte.

3.3 Das Jacobi-Relaxationsverfahren

Wir wollen nochmal die Iterationsvorschrift des Jacobi-Verfahrens betrachten:

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

Durch Umformung erhalten wir:

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

Wir addieren also zu u^k das Residuum. Die Idee ist nun das Residuum mit einem Parameter $\omega \in \mathbb{R}$ zu multiplizieren, so dass wir neue Eigenschaften erhalten:

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

Gleichung 3.18 stellt die Iterationsvorschrift für das Jacobi-Relaxationsverfahren dar. Die Iterationsmatrix ist 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 $i = 1, ..., n$:

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

Beachte: Für $\omega = 1$ erhalten wir das Jacobi-Verfahren. Der Rechenaufwand des Jacobi-Relaxationsverfahrens beträgt ebenfalls $\mathcal{O}(n^2)$.

3.3.2 Satz (Eigenwerte von $\mathbf{T}_{J_\omega}$ bzgl. $\mathbf{A}_{2D}$)

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

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

θ_i, θ_j wie in Unterabschnitt 2.3.1.

Beweis:

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

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

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

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

■

3.3.3 Lemma (Spektralradius von $\mathbf{T}_{J_\omega}$ bzgl. $\mathbf{A}_{2D}$)

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

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

Beweis:

Hier nutzen wir nochmals aus, dass $\sin^2(x) \in (0,1)$ monoton steigend ist für $x \in (0, \frac{\pi}{2})S$. Zudem ist $\cos(x) \in (0,1)$ für $x \in (0, \frac{\pi}{2})$ und $\sin^2(\frac{x}{2}) = \frac{1}{2}(1 - \cos(x))$. Für Gleichung 3.21 folgt also:

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

für $\omega \in (0,1)$.

■

Da mit $h = \frac{1}{m}$ und $m \in \mathbb{N}_{>2}$ für den Spektralradius des Jacobi-Verfahrens ($\omega = 1$) gilt:

$$\rho(\mathbf{T}) = \cos(\pi h) \in (0,1), \tag{3.23}$$

ist die Kondition der Matrix für $h \longrightarrow 0$ mit $\kappa_2(\mathbf{T}_J) = \frac{\lambda_{max}}{\lambda_{min}} \approx \frac{1}{\cos(\frac{\pi}{3})}$

$\cos(\pi h) \in [0,1)$ ist die Konditionszahl der Iterationsmatrix, die gegeben war durch:

$$\kappa_2(\mathbf{T}_{J_\omega}) = \frac{\lambda_{max}}{\lambda_{min}}, \tag{3.24}$$

für große m bzw. somit für kleine h

Für ein $\omega \in (0,1)$

Der Spektralradius ist für ein $\omega \in (0,1)$ näher an eins, als der des Jacobi-Verfahrens ($\omega = 1$). Das Jacobi-Relaxationsverfahren konvergiert also langsamer gegen die approximierte Lösung u^*, als das Jacobi-Verfahrens. Es wird daher nicht als iteratives Verfahren eingesetzt. Wie wir in Abschnitt 3.4 sehen werden, besitzt dieses Verfahren die Glättungseigenschaft und findet in den Mehrgitter-Algorithmen seine Anwendung. Für die zweidimensionale Poisson Gleichung stellen $\omega = \frac{1}{2}$ und $\omega = \frac{4}{5}$ die optimalen Parameter als Glätter dar [Saad].

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

Mit derselben Vorgehensweise wie in Unterabschnitt 3.2.5 verbessern wir noch den Rechenaufwand bzgl. $\mathbf{A}_{2D}$.

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

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^k + \frac{h^{-2}\omega}{4}(u_{i-1,j}^k + u_{i+1,j}^k + u_{i,j-1}^k + u_{i,j+1}^k) - \frac{\omega}{4}f_{i,j} \tag{3.25}$$

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

3.4 Glättungseigenschaft

Für die Iterationsmatrix des Jacobi-Relaxationsverfahrens gilt:

$$u^{k+1} = (Id - \omega D^{-1}A)u^k + \omega D^{-1}f. \tag{3.26}$$

Die Eigenvektoren sind gegeben durch:

$$(v_{k,l})_{i,j} = sin(i\theta_k)sin(j\theta_l) \text{ für } 1 \leq i, j, k, l \leq N, \tag{3.27}$$

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:

$$e^k = u^* - u^k. \tag{3.28}$$

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

$$\begin{aligned} e^{k+1} &= u^* - u^{k+1} = u^* - \Big((Id - \omega D^{-1}A)u^k + \omega D^{-1}f\Big) \\ &= \underbrace{u^* - u^k}_{=e^k} + \omega D^{-1}Au^k - \omega D^{-1}f = e^k + \omega D^{-1}(Au^k - f) \\ &= e^k + \omega D^{-1}(Au^k - Au^*) = e^k + \omega D^{-1}A(u^k - u^*) \\ &= e^k - \omega D^{-1}Ae^k = (Id - \omega D^{-1}A)e^k. \end{aligned} \tag{3.29}$$

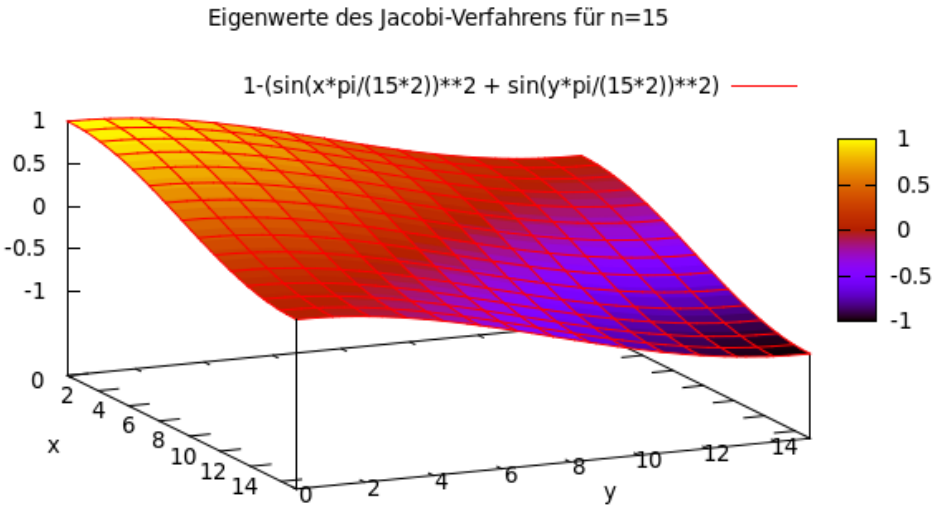

Abbildung 3.1: Dies ist das Eigenwertspektrum für $\omega = 1$. Das Gesamtschrittverfahren hat keine Glättungseigenschaft. Es liegen nur wenige der Eigenwerte um die Null.

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

der $v_{i,j}$ darstellen:

$$
\begin{aligned}
e^{k+1} &= \sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k+1} v_{i,j} = (Id - \omega D^{-1} A) \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} v_{i,j} \right) \\
&= \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} (Id - \omega \underbrace{D^{-1}}_{=\frac{1}{4}Id} A) v_{i,j} \right) = \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} (v_{i,j} - \frac{\omega}{4} \underbrace{A v_{i,j}}_{=\lambda_{i,j}(A) v_{i,j}}) \right) \\
&= \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} \underbrace{(1 - \frac{\omega}{4}\lambda_{i,j}(A))}_{=\lambda_{i,j}(\mathbf{T}_{J_\omega})} v_{i,j} \right) = \left(\sum_{i=1}^{N}\sum_{j=1}^{N} \xi_{i,j}^{k} \lambda_{i,j}(\mathbf{T}_{J_\omega}) v_{i,j} \right). \qquad (3.30)
\end{aligned}
$$

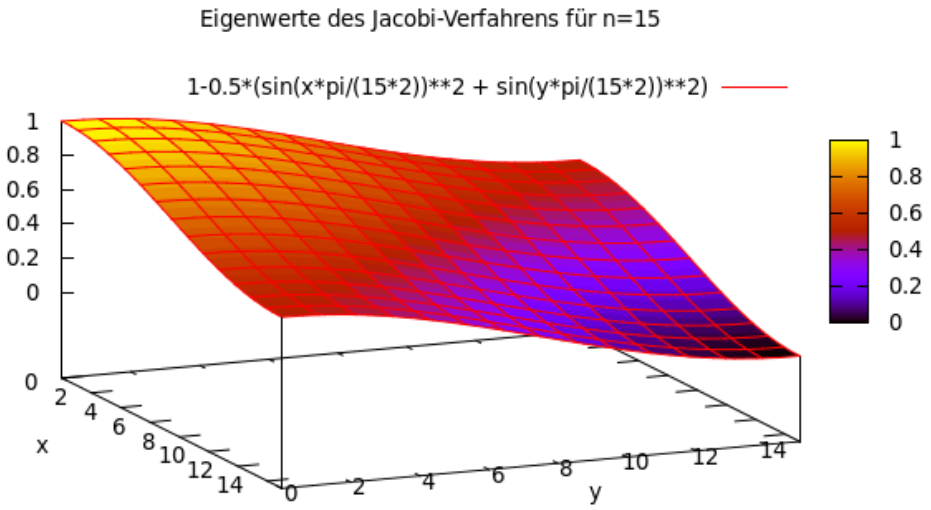

Abbildung 3.2: Für $\omega = \frac{1}{2}$ ist gut zu erkennen, dass für i, j nahe N viele Eigenwerte nahe Null liegen.

Da wir im Folgenden 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, d.h. zugehörige Eigenvektoren werden nahezu elimi-

niert. Diese Eigenschaft nennt man die *Glättungseigenschaft*. Wie die Abbildungen 3.4, 3.5 und 3.6 illustrieren, werden beim Glätten des Fehlers kurzwellige, stark oszillierende (hochfrequente) Fehleranteile nahezu ausgelöscht. Zwar heißt das nicht, dass der Fehler verschwindet oder kleiner wird, jedoch wird er glatt.
Es stellt sich z.B. heraus, dass der optimale Relaxationsparameter, der unabhängig von der Schrittweite h gewählt werden kann, $\omega = \frac{4}{5}$ ist [SAAD4]. In Abbildung 3.3 ist gut zu sehen, dass ein Großteil des Spektrums nahe Null liegt. 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). Diese liegen nahe 1 bzw. -1 und verändern die zugehörigen Eigenvektoren kaum. Als iteratives Verfahren ist $\omega = 1$ der optimale Parameter [**?**]. Das Jacobi-Verfahren besitzt wegen seines Eigenwertspektrums keine Glättungseigenschaft.

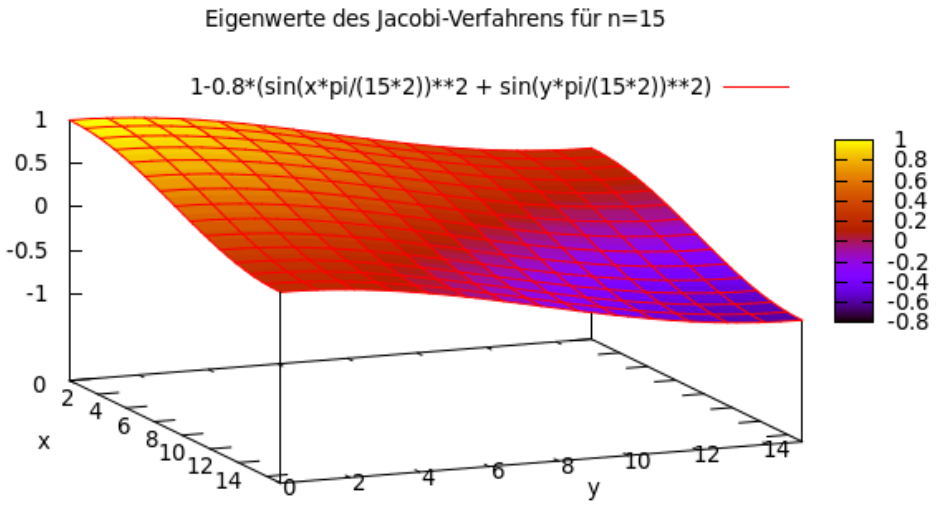

Abbildung 3.3: Für den optimalen Parameter $\omega = \frac{4}{5}$ ist gut zu erkennen, dass mehr Eigenwerte um die Null liegen, als beim Jacobi-Verfahren oder für den Parameter $\omega = \frac{1}{2}$ (orange Fläche) und somit eine gute Glättungseigenschaft besteht.

Zum Abschluss dieses Abschnitts wollen wir ein konkretes Beispiel zur Glättungseigenschaft anführen. Angenommen wir wählen unseren Startvektor $u^0 = \sin(\theta_i)$, mit $i = 1, ..., n$

und θ_i wie in Abschnitt 2.3. Außerdem sei die Poisson Gleichung aus Abschnitt 5.1 gegeben. Wir wählen den Parameter $\omega = \frac{4}{5}$ für das Jacobi-Relaxationsverfahren. Und wir berechnen den Fehlerterm $e^0 = u^* - u^0$ und plotten diesen. Der Fehler sieht zu Beginn wie folgt aus:

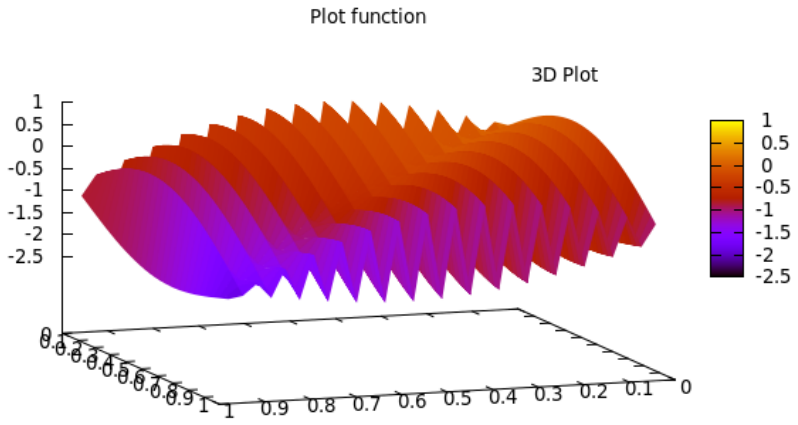

Abbildung 3.4: Die Oszillation der Sinuswellen im Startfehler ist hier gut zu erkennen.

Nun wollen wir uns noch die Plots nach einem und drei Iterationsschritten des Jacobi-Relaxationsverfahrens ansehen. Nach drei Schritten sind die starken Oszillationen verschwunden.

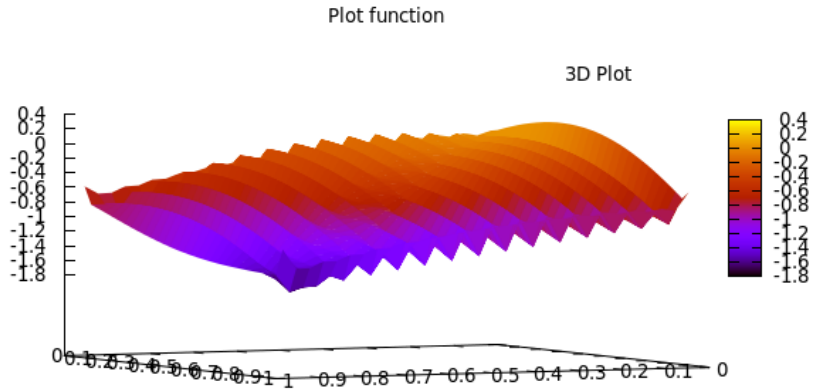

Abbildung 3.5: Nach einem Iterationsschritt ist bereits eine Glättung des Fehlers zu beobachten.

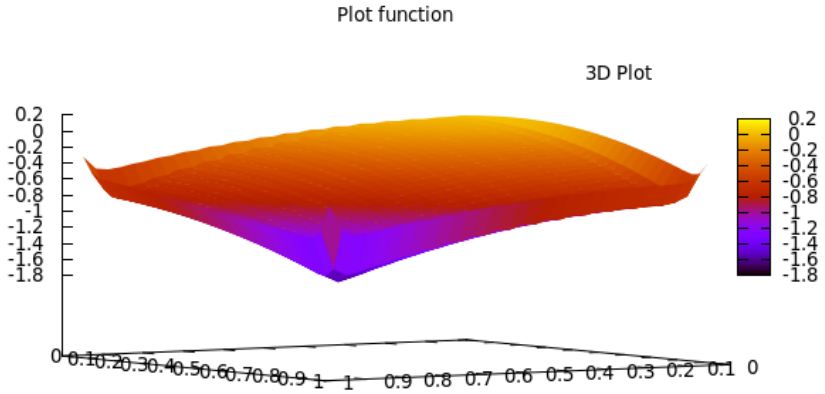

Abbildung 3.6: Man sieht, dass bereits nach 3 Iterationsschritten der Fehler glatt ist.

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, worauf wir allerdings im Folgenden nicht näher eingehen werden.

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.

Die A-Orthogonalität ist spezifisch für das CG-Verfahren. Bei der Aufstellung der Teilräume sind die zugehörigen Basisvektoren alle A-orthogonal, wie wir nach dem folgenden Satz sehen werden.

3.5.2 Satz (Minimierungsfunktion)

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

$$h(u) = \frac{1}{2}u^T Au - f^T u, \tag{3.31}$$

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

h hat ein eindeutig bestimmtes Minimum und

$$Au^* = f \Longleftrightarrow h(u^*) = \min_{u\in\mathbb{R}^n} h(u) \tag{3.32}$$

Einen kurzen Beweis hierzu findet man z.B. in [DR6].
Es ist also äquivalent, die Funktion $h(u)$ zu minimieren und das Gleichungssystem $Au = f$ zu lösen. Betrachtet man nun den Gradienten von $h(u)$, so gilt:

$$\nabla h(u) = Ax - f = -r. \tag{3.33}$$

Da wir bei diesem Verfahren stets das Minimum in einem Teilraum $U_k \in \mathbb{R}$ suchen, also die Funktion $h(u)$ minimieren wollen, 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 $\langle p^i, p^j \rangle_A = 0 \;\; f\ddot{u}r \;\; i \neq j$. Sei $v \in \mathbb{R}^n$, dann gilt für $u^k \in U_k$:

$$\|u^k - v\|_A = \min_{u \in U_k} \|u - v\|_A \tag{3.34}$$

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

$$P_{U_{k,\langle\cdot,\cdot\rangle}}(v) = \mathbf{u}^k = \sum_{j=0}^{k-1} \frac{\langle v, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j \tag{3.35}$$

Der Beweis zu diesem Lemma wird in [DR7] erläutert.

3.5.4 Allgemeiner Algorithmus der konjugierten Gradienten

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

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

$$U_1 = span\{r^0\}, \text{ wobei } r^0 = f - Au^0 \tag{3.36}$$

1. Bestimme eine A-orthogonale Basis

$$p^0, ..., p^{k-1} \text{ von } U_k. \tag{3.37}$$

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

$$\|u^k - u^*\|_A = \min_{u \in U_k} \|x - u^*\|_A. \tag{3.38}$$

Wir berechnen also:

$$u^k = \sum_{j=0}^{k-1} \frac{\langle u^*, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j. \tag{3.39}$$

3. Berechne erneut das Residuum und erweitere den Teilraum U_k

$$U_{k+1} = span\{p^0, ..., p^{k-1}, r^k\} \text{ wobei } r^k = f - Au^k. \tag{3.40}$$

Nachdem man ein Residuum berechnet hat, startet der erste Iterationsschritt. Man erweitert seinen Teilraum um das Residuum und bestimmt daraufhin 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 3.39 könnte man vermuten, dass die Lösung des Gleichungssystems 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 $y \in U_k$:

$$\langle u^*, y \rangle_A = \langle f, y \rangle \tag{3.41}$$

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 3.39 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:

$$u^k = u^{k-1} + \alpha_{k-1} p^{k-1}, \text{ mit } \alpha_{k-1} = \frac{\langle r^0, p^{k-1} \rangle}{\langle p^{k-1}, Ap^{k-1} \rangle}. \tag{3.42}$$

Für einen Beweis möchte ich auf [DR8] verweisen.

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:

$$r^k = r^{k-1} - \alpha_{k-1} Ap^{k-1}, \tag{3.43}$$

wobei α_{k-1} wie in Gleichung 3.42.

Beweis:

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

■

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

3.5.8 Satz (Bestimmung einer A-orthogonalen Basis)

Durch

$$p^{k-1} = r^{k-1} - \sum_{j=0}^{k-2} \frac{\langle r^{k-1}, p^j \rangle_A}{\langle p^j, p^j \rangle_A} p^j, \tag{3.44}$$

wird die A-orthogonale Basis des Unterraums (Krylow-Raums) $U_k = span\{p^0, ..., p^{k-2}, r^{k-1}\}$ 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 Herleitung und Beweis (siehe dazu beispielsweise [DR9]) angeben, wie man die p^k effizienter bestimmen kann.

3.5.9 Satz

Für die Berechnung von p^k gilt:

$$p^{k-1} = r^{k-1} - \frac{\langle r^{k-1}, Ap^{k-2} \rangle}{\langle p^{k-2}, Ap^{k-2} \rangle} p^{k-2}. \tag{3.45}$$

Substituiert man geschickt einige Werte in den Skalarprodukten (ohne Beweis), führt das auf den Algorithmus der konjugierten Gradienten.

3.5.10 Algorithmus der konjugierten Gradienten

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

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

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 und der Symmetrie - einen Aufwand von $\mathcal{O}(n)$ erreichen. Diesen erhält 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 zur Konvergenz findet man u.a. in [GL].

3.5.11 Satz (Konvergenz des CG-Algorithmus)

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 $Au = f$ und u^k die approximierte Lösung durch das CG-Verfahren ist. Dann gilt für $k = 1, 2, ...$:

$$
\|u^k - u^*\|_A \leq 2 \left(\frac{\kappa_2(A) - 1}{\kappa_2(A) + 1} \right)^k \|u^0 - u^*\|_A \tag{3.46}
$$

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.

Beispiel:

Für die zweidimensionale Poisson Matrix haben wir in Gleichung 2.36 gesehen, dass die Kondition der Matrix schlechter wird, je feiner das Gitter ist. Geht $h = \frac{1}{m}$ gegen Null, gilt

also $m \longrightarrow \infty$, so folgt:

$$\frac{\kappa(\mathbf{A}_{2D}) - 1}{\kappa(\mathbf{A}_{2D}) + 1} = 1 - \frac{2}{\kappa(\mathbf{A}_{2D}) + 1} \longrightarrow 1 \text{ für } \kappa(\mathbf{A}_{2D}) \longrightarrow \infty. \tag{3.47}$$

Für große m und mit Gleichung 3.46 sieht man, dass das CG-Verfahren nur langsam konvergiert. Das Ziel muss es also sein, eine Ausgangsmatrix für unser System zu finden, die besser konditioniert ist als $\mathbf{A}_{2D}$.

Bemerkung:

Das Verfahren der konjugierten Gradienten konvergiert im Allgemeinen nicht, wenn $\mathbf{A}$ 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 gezeigt 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 $\mathbf{A}$ durch eine ähnliche Matrix mit besserer Kondition zu ersetzen, damit sich das Konvergenzverhalten verbessert.

3.6.1 Satz

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

$$\mathbf{A}u = f \Longleftrightarrow \mathbf{W}^{-1}\mathbf{A}u = \mathbf{W}^{-1}f. \tag{3.48}$$

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

3.6.1.1 Beweis:

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

■

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, bei dem mit der Cholesky-Zerlegung eine entsprechende Umformung gefunden werden kann.
Die Formeln und Algorithmen zu den folgenden Cholesky-Zerlegungen gehen aus expliziten Formeln der LR-Zerlegung hervor. Da die Cholesky-Zerlegung eine LR-Zerlegung für symmetrisch positiv definite Matrizen darstellt, wollen wir diesen Aspekt nicht näher erläutern.

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 Lösung Mithilfe eines *beliebigen* Startvektors $u^0 \in \mathbb{R}^n$ zu einer gegebenen rechten Seite $f \in \mathbb{R}^n$. 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. Für eine schnelle Lösung von $\mathbf{W}z^k = r^k$ verwenden wir eine unvollständige Cholesky-Zerlegung.

3.6.3 Die unvollständige Cholesky-Zerlegung

Eine Matrix $\mathbf{A}$, die symmetrisch positiv definit ist, lässt sich durch eine Cholesky-Zerlegung in eine normierte untere Dreiecksmatrix $\mathbf{L}$ und eine rechte obere Dreicksmatrix $\mathbf{U}$ 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 $\mathbf{A}$ Nulleinträge besitzt, in L und U ebenfalls Null bleiben.

3.6.3.1 Definition (Muster E)

Ein Muster ist eine Teilmenge $E \subseteq \{1, ..., n\} \times \{1, ..., n\}$ für die gilt:

$$E = \{(i, j) | 1 \leq i, j \leq n, a_{i,j} \neq 0\}. \tag{3.49}$$

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

$$\mathbf{A} \approx \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T. \tag{3.50}$$

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: $\tilde{l}_{i,j} = 0$, 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 Algorithmus der unvollständigen Cholesky-Zerlegung

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

$$\begin{aligned} & \text{for } k = 1, ..., i-1 : \text{ if } (i,k) \in E, \\ \tilde{l}_{i,k} &= (a_{i,k} - \sum_{j=1}^{k-1} \tilde{l}_{i,j} \tilde{u}_{j,k}) / \tilde{u}_{k,k}; \\ & \text{for } k = i, ..., n : \text{ if } (i,k) \in E, \\ \tilde{r}_{i,k} &= a_{i,k} - \sum_{j=1}^{i-1} \tilde{l}_{i,j} \tilde{u}_{j,k}; \end{aligned}$$

Bemerkungen:

- Die für den PCG-Algorithmus wichtige Matrix $\mathbf{W}$ wird nun definiert 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 zur Vorkonditionierung. Wir wollen uns an dieser Stelle noch mit einer weiteren 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 $\hat{\mathbf{L}}$

Sei $e = (1, 1, ..., 1)^T$,

- $a_{i,j} = (\hat{\mathbf{L}}\hat{\mathbf{L}}^T)_{i,j}$ für alle $(i, j) \in E, i \neq j$,
- $\mathbf{A}e = \hat{\mathbf{L}}\hat{\mathbf{L}}^T e$, d.h. die Zeilensummen stimmen überein und
- $\hat{l}_{i,j} = \hat{r}_{i,j} = 0$ für alle $(i, j) \notin E$.

3.6.4.2 Algorithmus der modifizierten unvollständigen Cholesky-Zerlegung

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

$$
\begin{aligned}
& \text{drop} = 0; \\
& \text{for } k = 1, ..., i-1 : \\
& s = \sum_{j=1}^{k-1} \hat{l}_{i,j}\hat{u}_{j,k}; \\
& \text{if } (i,k) \in E : \hat{l}_{i,k} = (a_{i,k} - s)/\hat{u}_{k,k}; \\
& \text{else drop = drop + s;} \\
& \text{for } k = i, ..., n : \\
& s = \sum_{j=1}^{i-1} \hat{l}_{i,j}\hat{u}_{j,k}; \\
& \text{if } (i,k) \in E : \hat{r}_{i,k} = a_{i,k} - s; \\
& \text{else drop = drop + s;} \\
\hat{u}_{i,i} \;&=\; \hat{u}_{i,i} - drop; \qquad (3.51)
\end{aligned}
$$

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

Weitere Verfahren zur Vorkonditionierung, wie beispielsweise das SSOR-Verfahren, sind u.a. in [SAAD5] zu finden.

3.6.5 Effiziente Implementierung der modifizierten unvollständigen Cholesky-Zerlegungen

Gerade für diese Aufgabenstellung ist es von großem Interesse, wie der Algorithmus in Code umgesetzt wird. Da das Muster E unabhängig von der Dimension n ist haben beide Zerlegungen einen Rechenaufwand von $\mathcal{O}(n^2)$. Auch wenn wir nur über das Muster E iterieren, werden viele Werte der Matrix $\mathbf{A}$ überprüft. Dies kostet Rechenzeit.
Um dies zu optimieren, wird für $\mathbf{A}_{2D}$ eine weitere Matrix $\mathbf{B} \in \mathbb{R}^{n\times 5}$ eingeführt, die das Muster E ersetzt bzw. darstellt. Die 2D Poisson Matrix enthält maximal 5 Werte ungleich Null pro Zeile. Somit können in der $i-ten$ Zeile von $\mathbf{A}_{2D}$ maximal 5 Indizes j auftauchen, für die gilt $a_{i,j} \neq 0$. In $\mathbf{B}$ werden die Indizes j nacheinander abgespeichert. Sollten weniger als fünf Werte ungleich Null sein, so wird die zugehörige Zeile von $\mathbf{B}$ mit -1 aufgefüllt.

Beispiel:

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

$$B = \begin{pmatrix} 1 & 2 & (1+N) & -1 & -1 \\ 1 & 2 & 3 & (2+N) & -1 \\ & & \vdots & & \\ (5-N) & 4 & 5 & 6 & (5+N) \\ & & \vdots & & \\ (8-N) & 7 & 8 & 9 & -1 \\ (9-N) & 8 & 9 & -1 & -1 \end{pmatrix} \tag{3.52}$$

Mit der Matrix $\mathbf{B}$ 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::modifiedIncompleteCholesky(Matrix& A,WriteableMatrix& L ↩
    ,WriteableMatrix& U) {
2     int i,j,k,m,u,dim=A.Size();
3     double sum, drop;
4
```

```
5       for(i=0;i<dim;i++) {
6           drop=0;
7           for(k=0;k<5;k++) {
8               m=A.HashMatrix[i][k];
9               if(m!=-1 && m<i) {
10                  sum=0;
11                  for(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(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  }
```

Der Zugriff auf die richtigen Matrixeinträge von $\mathbf{A}_{2D}$ über $\mathbf{B}$ erfolgt hier mit die Methode $A.HashMatrix[i]k] = b_{i,k}$. Für $i = 1, ..., n$ und $k = 1, ..., 5$ wird eine Integer Variable $m = b_{i,k}$ gesetzt. In m wird nun für die $i - te$ Zeile der Matrix $\mathbf{A}_{2D}$ der $j - te$ Zeilenindex gespeichert, für den gilt: $a_{i,m} \neq 0$.

Der Rechenaufwand beträgt unter diesen Bedingungen lediglich $\mathcal{O}(5n) \approx \mathcal{O}(n)$.

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 vonstatten geht.

4 Mehrgitterverfahren

In diesem Abschnitt wollen wir uns mit den Mehrgittermethoden befassen. Wie Kapitel 3 gezeigt hat, spielt die Kondition der zugehörigen Matrix beim Lösen von Gleichungssystemen eine wichtige Rolle. Zwar konnten wir mit Vorkonditionierung diese Problem in den Griff bekommen, aber wie Kapitel 5 zeigen wird, wächst die Anzahl der Iterationsschritte auch bei den vorkonditionieren Verfahren mit der Gitterweite h mit. Mehrgitterverfahren hingegen sind Gitter unabhängig, d.h. die Anzahl der Iterationsschritte (oder Zyklen) hängt nicht davon ab, wie fein wir unser Gitter wählen ([SAAD6]). Der Name für diese Verfahren leitet sich dauraus ab, dass wir gewisse Operationen nicht nur auf dem zugrunde liegenden Gitter ausführen, sondern auf gröbere Gitter wechseln. Dadurch erhalten wir mehrere Gitter beim Lösen des Gleichungssystems.
Die Idee dieser Methoden beruht auf der Fehlerglättung. Wir haben bereits festgestellt, dass beispielsweise das Jacobi-Relaxationsverfahren kurzwellige Fehler nach wenigen Iterationsschritten nahezu auslöscht. Zudem werden die niederfrequenten Fehleranteile auf dem feinen Gitter zu hochfrequenten Fehlern auf dem groben Gitter (siehe dazu beispielsweise [STR]).
Wir wollen uns in diesem Kapitel an [SAAD7] halten. Spezifische Matrizen für das zweidimensionale Poisson Problem wurden selbst aufgestellt.

4.1 Grundlagen

1. Glättungseigenschaft der Jacobi-Relaxation
   Die kurzwelligen Fehlerterme werden nach wenigen Iterationsschritten geglättet bzw. verschwinden nahezu (siehe Abschnitt 3.4).

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 als

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

wobei u^* die exakte Lösung darstellt. Betrachten wir jetzt nochmals Gleichung 3.7:

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

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 bilineare Interpolation.

Bemerkung:

Wir wählen der Einfachheit halber das N stets so, dass gilt: $N = 2^n - 1$, wobei $n \in \mathbb{N}$. Somit können wir die Schrittweite h auf jedem Gitter komfortabel bestimmen. Es gilt dann z.B. für $N_h = 2^n - 1$ und für $N_{2h} = 2^{n-1} - 1$.

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 inneren Gitterpunkte in x- und y-Richtung auf dem groben Gitter sind. Dabei überführt die Matrix $\mathbf{I}$ Vektoren von Ω_{2h} auf Ω_h. Sie ist *nicht* quadratisch und kann verschiedene Gestalten haben,z.B.:

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

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

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

Wir haben hier die Full-Weighting-Matrix der Interpolation verwendet, da sie zu einem besseren Ergebnis führt. 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-Weighting-Operators dient Abbildung 4.1. Es geht in jedes $u_h^{i,j}$ auch der gewichtete Wert aller Nachbarpunkte von $u_{2h}^{i,j}$ des groben Gitters mit ein.

Bemerkung:

Für den Fall von eindimensionalen Gittern hätte $\mathbf{I}$ folgende Darstellung:

$$\mathbf{I} = \frac{1}{2} \begin{pmatrix} 1 & & & & & & \\ 2 & & & & & & \\ 1 & 1 & & & & & \\ & 2 & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & & 1 & 1 & \\ & & & & & 2 & \\ & & & & & 1 & \end{pmatrix}. \tag{4.8}$$

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

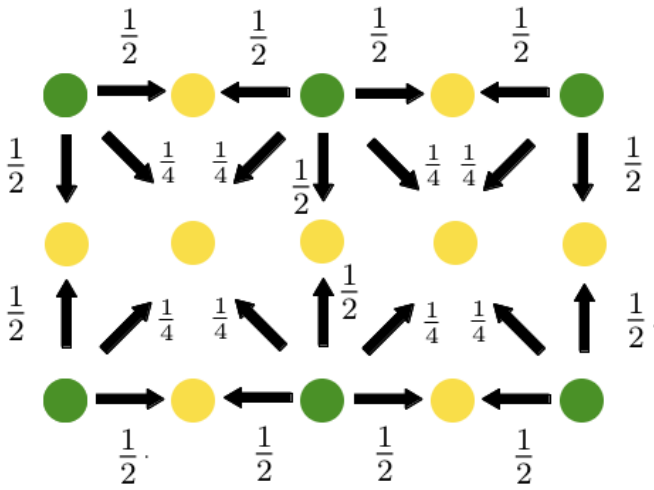

Abbildung 4.1: Bei der Interpolation bedienen sich die Punkte vom feinen Gitter (grün und gelb), bei den gewichteten Punkten des groben Gitters (grün). Direkte Nachbarn werden mit $\frac{1}{2}$, diagonale Nachbarn mit $\frac{1}{4}$ gewichtet.

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 geben wir die Interpolation in Komponentenschreibweise an:

$$u_h^{2i-1,2j-1} = u_{2h}^{i,j}\ i,j = 1,...,\tilde{N}, \tag{4.9}$$

$$u_h^{2i-1,2j} = \frac{1}{2}(u_{2h}^{i,j} + u_{2h}^{i,j+1})\ i = 1,...,\tilde{N}; j = 1,...,\tilde{N}-1, \tag{4.10}$$

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

$$u_h^{2i,2j} = \frac{1}{4}(u_{2h}^{i,j} + u_{2h}^{i,j+1} + u_{2h}^{i+1,j} + u_{2h}^{i+1,j+1})\ i,j = 1,...,\tilde{N}-1. \tag{4.12}$$

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**

$$\mathbf{R} = \frac{1}{4}\mathbf{I}^T \tag{4.13}$$

4.3.1 Restriktionsmatrix

Dadurch ist die Matrixdarstellung gegeben durch:

$$R = \frac{1}{16}\begin{pmatrix} I_1^T & I_2^T & & & \\ & I_1^T & I_2^T & & \\ & & & \ddots & \\ & & & I_1^T & I_2^T \end{pmatrix}, \tag{4.14}$$

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

Für die Umsetzung in Code benutzen wir die Komponentenschreibweise:

Für ein u_{2h} auf Ω_{2h} gilt:

$$\begin{aligned} u_{2h}^{i,j} &= \frac{1}{16}(4u_h^{i,j} + 2(u_h^{i+1,j} + u_h^{i-1,j} + u_h^{i,j+1} + u_h^{i,j-1}) \\ &+ u_h^{i-1,j-1} + u_h^{i-1,j+1} + u_h^{i+1,j+1} + u_h^{i+1,j-1}) \end{aligned} \tag{4.15}$$

Das Vorgehen wird in Abbildung 4.2 illustriert.

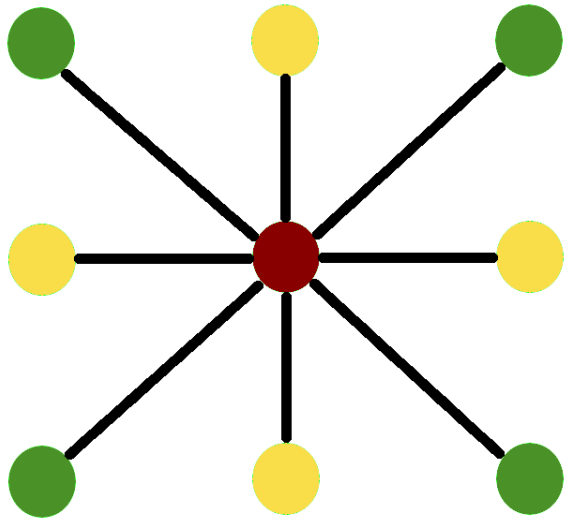

Abbildung 4.2: Ausgehend von einem Punkt innerhalb des Gitters, bedient dieser sich der Werte seiner direkten ($\frac{2}{16}$ in gelb) und diagonalen Nachbarn ($\frac{1}{16}$ in grün). Der Punkt, in dem restringiert wird fließt mit einer Gewichtung von $\frac{4}{16}$ (rot) in den neuen Wert ein.

Bemerkung:

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

$$\mathbf{R} = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & \\ & 1 & 2 & 1 & & & \\ & & & \ddots & & & \\ & & & & 1 & 2 & 1 \end{pmatrix} \tag{4.16}$$

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:

$$\mathbf{A}_{2h} = \mathbf{R}\mathbf{A}_h\mathbf{I}, \tag{4.17}$$

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 diesen Abschnitt 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 inneren Gitterpunkte in x- und y-Richtung des feinen Gitters und $\tilde{N} = 3$ die Anzahl der Gitterpunkte in x- bzw. y-Richtung 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:

$$\mathbf{R}\mathbf{A}_{2D_h}\mathbf{I} = \mathbf{R}\frac{1}{h^2}\begin{pmatrix} A_1 & -Id & & \\ -Id & A_2 & \ddots & \\ & \ddots & \ddots & \\ & & A_6 & -Id \\ & & -Id & A_7 \end{pmatrix}\mathbf{I} = \frac{1}{(2h)^2}\begin{pmatrix} A_1 & -Id & \\ -Id & A_2 & -Id \\ & -Id & A_3 \end{pmatrix} = \mathbf{A}_{2D_{2h}} \tag{4.18}$$

Die Matrizen $\mathbf{A}_{2D_{ih}}$ sind also für alle $i \in \{2^n \in \mathbb{N} | n \in \mathbb{N}\}$ stets bekannt.

4.5 Das Zweigitterverfahren

Wie in Abschnitt 2.2 gesehen befinden wir uns bei der Diskretisierung der Poisson-Gleichung auf einem Gebiet $\Omega_h = (0,1)^2$ mit Schrittweite $h = \frac{1}{m}$. Nach der Ausführung

von $\tilde{k}$ Iterationsschritten des Jacobi-Relaxationsverfahren sind auf diesem Gitter die kurzwelligen Fehler $e^k = u^* - u^k$ geglättet bzw. nahezu verschwunden. Zudem haben wir die Äquivalenz der Gleichung $\mathbf{A}u = f$ und $\mathbf{A}e = r$ für $e = 0$ gesehen.
Wir berechnen nach $\tilde{k}$ Glättungsschritten das Residuum r^k und wollen sodann $\mathbf{A}_h e_h^k = r_h^k$ lösen. Die Lösung dieser Gleichung auf dem einem gröberen Gitter ist natürlich wesentlich günstiger, als auf dem feinen Gitter. Darum bringen wir r_h^k durch Restriktion auf das gröbere Gitter und lösen die Residuumsgleichung dort direkt. Anschließend bringen wir den approximierten Fehlerterem e_{2h}^k durch Prolongation zurück auf das feine Gitter und addieren e_h^k zu u_h^k, da $u^* = u^k + e^k$. Abschließend wird $\tilde{k}$ mal nachgeglättet. Wir wiederholen dieses Vorgehen bis zur Konvergenz. Nachfolgend der Pseudocode für den Algorithmus des Zweigitterverfahrens:

while	$\|\|u^k - u^*\\| < TOL$	
	pre-smooth	JacobiRelaxationMethod
	calculate residual	$r^k = b - Au^k$
	restrict	$r_{2h}^k = Rr_h^k$
		$\mathbf{A}^{2h} = R\mathbf{A}^h I$
	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 schuldig bleiben. Für ein weiteres Studium wird [SAAD8] empfohlen. Als Löser verwenden wir

den PCG-Algorithmus aus Kapitel 3 mit einer modifizierten unvollständigen Cholesky-Zerlegung.

Der klare Nachteil dieser Methode liegt natürlich am 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 $n = 127^2$. Ein System dieser Ordnung zu lösen erfordert großen Rechenaufwand, der in dieser Form nicht immer erwünscht ist.

4.6 Mehrgitter-Algorithmen

Da also das Lösen der Residuumsgleichung auf dem groben Gitter einen Löser erfordert, der zusätzlichen Rechenaufwand bedeutet, ist der Zweigitter-Algorithmus nicht immer die erste Wahl zum Lösen eines Gleichungssystems.

Eine andere Methode ist, das Gitter immer gröber zu machen, bis das System direkt lösbar ist oder zumindest entsprechend klein 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, sobald sie sich 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

Die Schritte sind im wesentlichen die gleichen, als die beim Zweigitterverfahren. Es wird auf jedem Gitter eine a priori Glättung durchgeführt und das Residuum restringiert. Auf dem gröbsten Gitter wird die Residuumsgleichung mit $e = 0$ gelöst. Der Fehlerterm wird anschließend prolongiert und auf dem nächst feineren Gitter zum jeweiligen u^k dazu addiert. Nach einer a posteriori Glättung wird das neu berechnete u^k ebenfalls prolongiert. Sobald wir auf das feinste Gitter zurück gekehrt sind, erhalten wir eine neue Approximation für u_h^k.

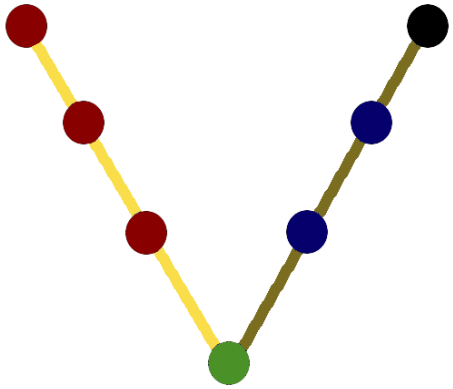

Abbildung 4.3: rot: a priori Glättung, **gelb:** Restriktion, **grün:** direktes Lösen, **braun:** Prolongation, **blau:** a posteriori Glättun und **schwarz:** a posteriori Glättung und Korrektur des Fehlers.

Dieser Algorithmus ist auch als V-Zyklus bekannt. Wie dieser Name zustande kommt, illustriert Abbildung 4.3. Nun gibt es eine weitere Variante, den W-Zyklus (illustriert in Abbildung 4.4). Durch zweifachen rekursiven Aufruf der Funktion entstehen wesentlich mehr Wechsel zwischen den Gittern. Natürlich ist die Rechenzeit durch häufigeres glätten, lösen, restringieren und prolongieren höher:

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	$r_{2h}^k = Rr_h^k$
	$\mathbf{A}^{2h} = R\mathbf{A}^h P$
recursion	$\tilde{e}_{2h}^k = \textbf{W-cycle}(0, r_{2h}^k)$
recursion	$e_{2h}^k = \textbf{W-cycle}(0, \tilde{e}_{2h}^k)$
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

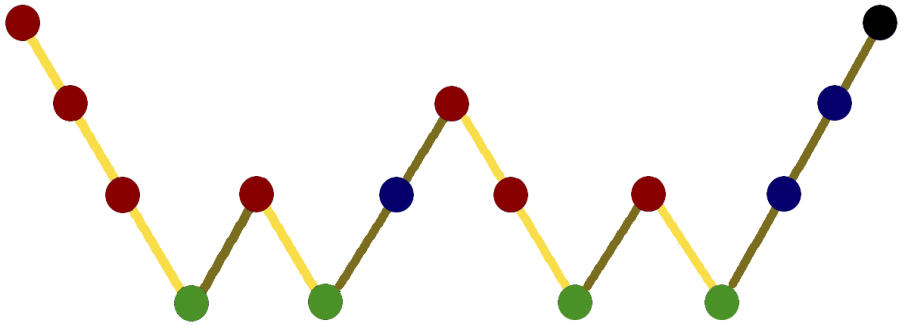

Abbildung 4.4: **rot:** a priori Glättung, **gelb:** Restriktion, **grün:** direktes Lösen, **braun:** Prolongation, **blau:** a posteriori Glättun und **schwarz:** a posteriori Glättung und Korrektur des Fehlers.

Für manche Systeme erhält man durch schnellere Konvergenz eine kürzere Rechenzeit, als bei einem V-Zyklus. Im Falle der Poisson Gleichung ist dies nicht der Fall, wie Kapitel 5 zei-

gen wird. Außerdem ist dort die Gitterunabhängigkeit schön zu sehen, da die Anzahl der benötigten Zyklen bis zur Konvergenz nahezu konstant ist.

5 Implementierung und Beispiel

In diesem letzten Kapitel werden praktische Code-Beispiele gezeigt. Außerdem werden anhand eines Beispiels die Iterationsschritte und Rechenzeit der verschiedenen Verfahren betrachtet.

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

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

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

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 folgendem Graphen ergeben:

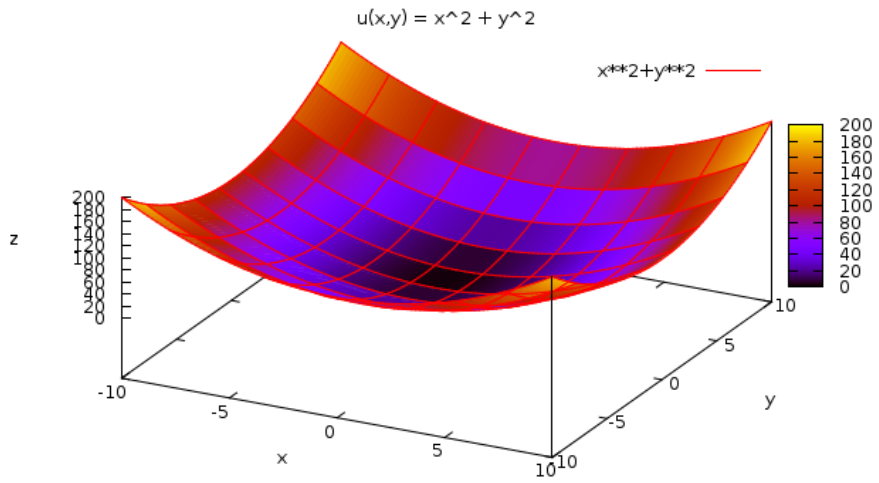

Abbildung 5.1: Die analytische Lösung für diese Poisson-Gleichung war gegeben durch $u(x, y) = x^2 + y^2$.

Zu beachten ist, dass wir $\Omega_h \in (0, 1)^2$ gewählt haben. Die Lösung unseres Systems sollte folgenden Graphen als Ergebnis haben:

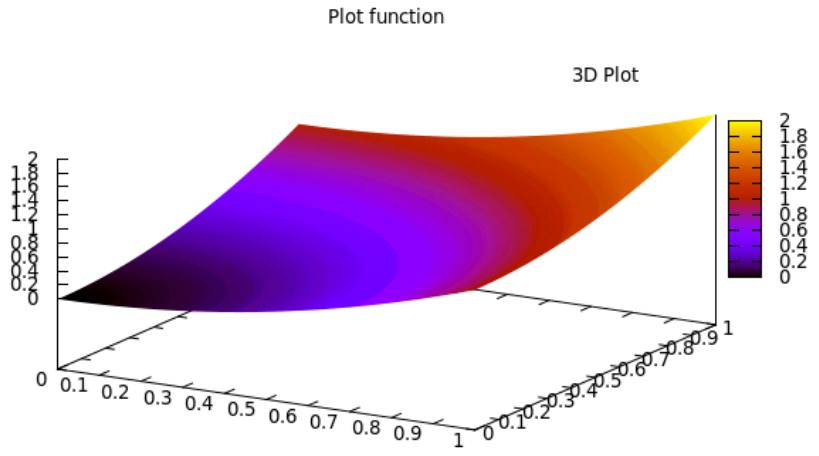

Abbildung 5.2: Dieser Graph entspricht der Lösung der Poisson-Gleichung auf dem Einheitsquadrat mit $h = \frac{1}{1024}$. Als Ausgangsdaten wurde die Lösung durch einen Mehrgitter-V-Zyklus mit 3 Gittern verwendet mit $m = 1024$.

5.2 Bemerkung 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 die Lösung der Poisson Gleichung für verschiedene Verfahren betrachten. Dafür werden einige der Methoden dargestellt. Wir wollen 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 Speicherung von $\mathbf{A}_{2D}$

Für eine effiziente Implementierung ist die Speicherung der Poisson-Matrix ein wichtiges Kriterium. Da die Werte der Diagonale und die der Nebendiagonalen zu jeder Zeit bekannt sind und die Matrix symmetrisch ist, werden in der C++-Methode PoissonMatrix lediglich drei double-Werte gespeichert. Somit benötigen wir lediglich drei mal 8 Byte, also 24 Byte, Speicherplatz für die gesamte Matrix.
Für die Matrizen **L** und **R**, deren Werte ebenfalls nur auf der Diagonale und den 4 Nebendiagonalen ungleich Null sind und die beide ebenfalls symmetrisch sind, gehen wir ähnlich vor. Da dort die Werte jedoch unterschiedlich sind, werden pro Matrix drei Vektoren (vector<double>) angelegt, die pro Matrix einen Speicherplatz von $< 24n$ Bytes ausmachen. Hier hängt zwar der Speicherplatz von der Größe der Matrix ab, ist aber immer noch überschaubar.
Durch diese Art der Speicherung können wir die Matrix-Vektor-Multiplikation, die im Allgemeinen einen Rechenaufwand von $\mathcal{O}(n^2)$ hat, anpassen, so dass der Aufwand auf $\mathcal{O}(n)$ sinkt.

```
1 vector<double> PoissonMatrix::operator*(const vector<double>& x) {
2     int dim=x.size(),n=sqrt(dim);
3     vector<double> tmp(dim,0);
4     for(int i=0;i<dim;i++) {
5         tmp[i]+=x[i]*4.0*pow(n+1,2);
6         if(i<(dim-n)) {
7             tmp[i]+=x[i+n]*-1.0*pow(n+1,2);
8             tmp[i+n]+=x[i]*-1.0*pow(n+1,2);
```

```
9           }
10          if(i%n!=0) {
11              tmp[i]+=x[i-1]*-1.0*pow(n+1,2);
12              tmp[i-1]+=x[i]*-1.0*pow(n+1,2);
13          }
14      }
15      return tmp;
16  }
```

5.4 Abbruchkriterien

Um zu verstehen, warum wir Abbruchkriterien benötigen, hier ein kurzes Beispiel:
Das CG-Verfahren konvergiert nach maximal n Schritten bei exakter Arithmetik. Wir bräuchten somit 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, nämlich gerade n. Um dies zu vermeiden, lässt man den Algorithmus abbrechen, sobald eine gewisse Toleranzgrenze erreicht ist. In der Praxis schätzt man beispielsweise das $k - te$ Residuum in der A-Norm oder der 2-Norm gegen r^0 ab. In diesem Beispiel wählen wir folgenden Ansatz:

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

wobei $u^* \in \mathbb{R}^n$ die Lösung des Gleichungssystems darstellt.
Im Allgemeinen ist natürlich die Lösung der partiellen Differentialgleichung nicht bekannt. Hier existiert die analytische Lösung und wir können das Abbruchkriterium so wählen.

Bemerkung:

In [DR4] findet man für einige dieser Verfahren und dem gewählten Abbruchkriterium Tabellen mit Vergleichswerten für $m = 40, 80, 160, 320$. Da wir die Werte für diese m bereits kennen, beschränken wir uns auf Werte für $m = 2^n - 1, n \in \mathbb{N}_{>0}$, um einen besseren Vergleich zu den Mehrgitterverfahren zu erhalten.

5.5 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 5.1.

m	32	64	128	256
Iterationsschritte	1340	5344	21341	85282
Rechenzeit in s	0.17	2.89	47.22	771.03

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

Es ist gut zu sehen, dass sich die Anzahl der Iterationsschritte vervierfacht, sobald m verdoppelt wird.

5.6 Jacobi-Relaxations-Verfahren

Zunächst soll der C++-Code angegeben werden. Setzt man $\omega = 1$ erhält man obiges Jacobi-Verfahrens.

```
1  int Algorithms::JacobiRelaxationMethod(Matrix& A,vector<double>& x,const
       vector<double>& b,const vector<double>& solved) {
2      vector<double> tmp;
3      double sum,omega=4.0/5.0;
4      int n=sqrt(x.size()),dim=n*n,steps=0;
5      double h=1.0/(double)(n+1);
6      vector<double> r(x.size());
7      r=x-solved;
8      double TOL=(r|r)*pow(10,-3);
9      while(TOL<=(r|r)) {
10         tmp=x;
11         for(int i=0;i<n*n;i++) {
12             sum=0.0;
13             if(i>=n && i<dim-n) sum+=tmp[i-n]+tmp[i+n];
14             if(i<n) sum+=tmp[i+n];
```

```
15              if(i>=dim-n) sum+=tmp[i-n];
16              if(i%n!=0) sum+=tmp[i-1];
17              if(i%n!=n-1) sum+=tmp[i+1];
18              x[i]=omega*1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum)+
                    tmp[i]*(1-omega);
19              r[i]=x[i]-solved[i];
20          }
21          steps++;
22      }
23      return steps;
24  }
```

Der Code lässt gut erkennen, wie die Struktur von $\mathbf{A}_{2D}$ gut ausgenutzt werden kann.

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

In Unterabschnitt 3.3.3 haben wir gesehen, das der Spektralradius des Jacobi-Relaxationsverfahrens näher an eins liegt als der des Jacobi-Verfahrens, denn:

$$\cos(\pi h) < 1 - \omega(1 - \cos(\pi h)) < 1, \tag{5.4}$$

für $\omega \in (0, 1)$.
Aus diesem Grund benötigt das relaxierte Verfahren auch wesentlich mehr Iterationsschritte. Wir betrachten das Verfahren, für den Parameter $\omega = \frac{4}{5}$.

m	32	64	128	256
Iterationsschritte	1676	6681	26676	106603
Rechenzeit in s	0.23	3.86	62.95	1031.14

Tabelle 5.2: Es ist deutlich zu erkennen, dass das Jacobi-Relaxationsverfahrens als iterativer Löser ungeeignet ist.

5.7 CG-Verfahren

Da das CG-Verfahren eines der effizienteren Iterationsverfahren ist, sollten die Messwerte entsprechend gut sein. Man erkennt aber in Tabelle 5.3 die schlechte Kondition von $\mathbf{A}_{2D}$. Pro Verdopplung der Gitterweite, verdoppeln sich auch die Iterationsschritte. Die Rechenzeit für ein sehr feines Gitter ist ebenfalls nicht akzeptabel.

m	32	64	128	256	512
Iterationsschritte	52	104	210	420	841
Rechenzeit in s	0.02	0.15	1.33	10.96	87.76

Tabelle 5.3: Ein gutes iteratives Verfahren, dass jedoch für feine Gitter nicht die gewünschte Effizienz aufweist.

Der C++-Code für das CG-Verfahren, wie auch im nächsten Abschnitt für das PCG-Verfahren ist verhältnismäßig lang. Wir wollen daher auf das Codebeispiel verzichten.

5.8 PCG-Verfahren

5.8.1 Mit unvollständiger Cholesky-Zerlegung

Auch wenn man eine klare Verbesserung zum Standardverfahren der konjugierten Gradienten sieht, sind die Ergebnisse immer noch nicht in einem akzeptablen Rahmen.

m	32	64	128	256	512	1024
Iterationsschritte	16	32	63	126	251	502
Rechenzeit in s	0.01	0.10	0.74	5.97	47.09	373.97

Tabelle 5.4: Es ist anhand der Tabelle schön zu erkennen, dass die Anzahl der Schritte proportional mit der Gitterweite zunimmt.

Offensichtlich gilt für das vorkonditionierte Verfahren der konjugierten Gradienten, dass die Anzahl der Schritte ungefähr halb so groß ist, als die Gitterweite m. Für die feinsten Gitter ist die Rechenzeit allerdings ungenügend.

5.8.2 Mit modifizierter unvollständiger Cholesky-Zerlegung

Im Fall der modifizierten unvollständigen Cholesky-Zerlegung erwartet man nun einen deutlichen Effekt auf Anzahl der Iterationsschritte und Rechenzeit.

m	32	64	128	256	512	1024
Iterationsschritte	7	9	12	17	24	33
Rechenzeit in s	0.01	0.03	0.17	0.96	5.12	27.07

Tabelle 5.5: Es ist deutlich zu erkennen, dass diese Variante des PCG-Verfahrens die effizientere von beiden darstellt.

Wie Tabelle 5.5 nun eindrucksvoll zeigt, brauchen wir für das feinster Gitter maximal 33 Iterationsschritte. Auch die Rechenzeit ist mit 27.07 Sekunden gut. Wie wir im nächsten Abschnitt sehen werden, sind lediglich die Mehrgittermethoden schneller und effizienter.

5.9 Das Mehrgitterverfahren

Auch wenn der Zweigitteralgorithmus nicht immer die beste Wahl zur Lösung einer partiellen Differentialgleichung ist, stellt er speziell für die Poisson Gleichung eine äußerst effiziente Methode dar. Besonders bemerkenswert ist bei den Mehrgitterverfahren die Feinheit des Gitters, dass mit einer maximalen Schrittweite von $h = \frac{1}{4096}$ diskretisiert werden kann.

5.9.1 Zweigitterverfahren

Als Löser auf dem gröbsten Gitter verwenden wir das PCG-Verfahren mit einer modifizierten unvollständigen Cholesky-Zerlegung verwendet.

m	32	64	128	256	512	1024	2048	4096
Anzahl der Zyklen	3	3	3	3	3	3	3	3
Rechenzeit in s	0.01	0.02	0.13	0.55	2.54	12.08	57.66	282.13

Tabelle 5.6: Das Lösen auf dem groben Gitter erfolgt mit der modifizierten unvollständigen Cholesky-Zerlegung.

Ein Iterationsschritt ist ein Zyklus vom feinen auf das grobe Gitter und zurück.

5.9.2 V-Zyklus

Betrachten wir nun mehr als zwei Gitter, stellt sich durch "trial and error"heraus, dass für die Poisson-Gleichung ein 3-Gitter-V-Zyklus die günstigste Variante darstellt. Nachfolgend sind die durchaus imposanten Werte dargestellt:

m	32	64	128	256	512	1024	2048	4096
Anzahl der Zyklen	5	4	4	4	4	4	4	4
Rechenzeit in s	0.00	0.02	0.01	0.47	1.97	8.30	35.44	154.53

Tabelle 5.7: Die Messwerte für einen V-Zyklus mit einem feinen und zwei groben Gittern.

Wie nun deutlich wird, sind beim Zweigitterverfahren, wie auch bei der Verwendung eines V-Zyklus, die Anzahl der Zyklen nahezu konstant. Es ist also nicht vom Gitter abhängig, wie oft ein Zyklus durchlaufen werden muss, damit der Algorithmus konvergiert.
Zwar benötigen wir beim V-Zyklus ca. einen Durchlauf mehr, dafür hat dieser eine geringer Rechenzeit, da er auf einem noch gröberen Gitter lösen kann.
Vergleichen wir nun die Mehrgittermethoden, insbesondere V-Zyklen, mit dem PCG-Verfahren, so sind deutliche Unterschiede zu erkennen. Das schnellere der vorkonditionieren CG-Verfahren war mit einer modifizierten unvollständigen Cholesky-Zerlegung. Aber auch bei diesem Verfahren, ist die Anzahl der Iterationsschritte mit kleinerer Gitterweite h gewachsen. Auch die Rechenzeit liegt für ein Gitter mit $m = 1024$ weit über denen der Mehrgittermethoden. Bei diesen war es durch ihre Gitterunabhängigkeit sogar möglich, ein noch feineres m als dieses zu wählen.

5.9.3 W-Zyklus

Abschließend werfen wir noch einen Blick auf einen W-Zyklus. Auch hier stellt sich heraus, dass drei Gitter am günstigsten sind.

m	32	64	128	256	512	1024	2048	4096
Anzahl der Zyklen	5	5	5	4	4	4	4	3
Rechenzeit in s	0.01	0.05	0.22	0.82	3.74	17.31	81.88	297.59

Tabelle 5.8: Auch bei einem W-Zyklus sieht man deutlich, dass die Anzahl der benötigten Zyklen unabhängig von der Gitterweite ist.

Zwar braucht der W-Zyklus nicht mehr Iterationsschritte, jedoch ist der Rechenaufwand höher. Es werden pro Zyklus mehr Restriktionen und Interpolationen benötigt. Zudem werden auf jedem Gitter Glättungen durchgeführt. Und es wird öfter direkt gelöst. Dies benötigt mehr Rechenzeit, was durch die Tabelle belegt wird.

Die Implementierung der drei Mehrgittermethoden ist die gleiche. Es wird lediglich beim Aufruf der C++-Methode die Anzahl der Gitter mitgegeben und ob es ein V- oder W-Zyklus sein soll. Wählt man zwei Gitter, so fungiert der Algorithmus als Zweigitterverfahren. Dieser sieht dann wie folgt aus:

5.9.4 Mehrgitteralgorithmus C++-Methode

```
1  vector<double> Algorithms::Cycle(Matrix& A,vector<double>& x,const
       vector<double>& b,int lambda,int theta,Matrix& B,WriteableMatrix& L,
       WriteableMatrix& U) {
2       int dim=x.size(),n=sqrt(dim),N2h=(n+1)/2-1,dim2h=pow(N2h,2);
3       vector<double> r(dim,0),E(dim,0),r2h(dim2h,0),E2h(dim2h,0);
4       if(this->Vcounter==lambda) {
5           PCGdirect(B,L,U,x,b);
6           return x;
7       } else {
8           this->Vcounter++;
9           JacobiRelaxation(A,x,b,3);
10          r=b-A*x;
```

```
11          Restriction(r,r2h,n);
12          E2h=Cycle(A,E2h,r2h,lambda,theta,B,L,U);
13          if(theta==1) E2h=Cycle(A,E2h,r2h,lambda,theta,B,L,U);
14          Interpolation(E2h,E,n);
15          x+=E;
16          JacobiRelaxation(A,x,b,3);
17          this->Vcounter--;
18          return x;
19      }
20  }
```

6 Fazit

Diese Arbeit hat gezeigt, wie die Poisson Gleichung auf ihrem Gebiet diskretisiert werden kann. Das daraus resultierende Gleichungssystem $\mathbf{A}u = f$ ermöglicht uns auf eine Bandbreite von Lösungsverfahren zurück greifen zu können. Wir haben gesehen, dass iterative Verfahren ihre Daseinsberechtigung haben, da einige von ihnen schnell sehr gute Lösungen des Gleichungssystems liefern. Allen voran ist hier das Verfahren der konjugierten Gradienten. Konditioniert man die Ausgangsmatrix geschickt vor, so können wir große Systeme effizient lösen. Das Umsetzen in Programmcode ist für die vorkonditionieren Verfahren nicht immer einfach, vor allem weil man versuchen möchte, die Rechenzeit so klein wie möglich zu halten.
Die Implementierung der Mehrgittermethoden dagegen ist weitaus einfacher. Jedoch ist hier darauf zu achten, wie man die Poisson Gleichung aufstellt und wie Operationen wie Restriktion und Prolongation geschickt programmiert werden. Hat man ein fertigs Programm, so wird man mit einer Gitter unabhängigen Anzahl an Zyklen und enorm kurzer Rechenzeit belohnt.
Die Mehrgittermethoden stellen zur Lösung der Poisson Gleichung ein hervorragendes Werkzeug dar. Es ist schwer vorstellbar, dass in naher Zukunft ein ähnlich effizienter Algorithmus entwickelt wird, um die Lösung dieser Gleichung zu ermöglichen.

7 C++-Code

7.1 Hauptprogramm

```
#include "classes.h"

int main(int argc, char const *argv[]) {
    const char *method;
    int arg,alg,steps=0;
    if(argc<4) {
        printf("\n");
        printf("Choose your prefered algorithm:\n[1] Conjugate Gradient\
            n[2] P. Conjugate Gradient (ICG)\n[3] P. Conjugate Gradient (
            MICG)\n[4] Two Grid (square numbers of 2 recommended!)\n[5] V
            -Cycle (square numbers of 2 recommended!)\n[6] W-Cycle (
            square numbers of 2 recommended!)\n[7] Jacobi Method\n[8] 
            Jacobi Relaxation Method \n[9] Gauss-Seidel-Method\n[10] SOR 
            Method\n");
        scanf("%d",&alg);
        printf("Choose your m, where h=1/m and N=m-1 number of grid 
            points in x and y direction:\n");
        scanf("%d",&arg);
        arg=arg-1;
    } else {
      arg=atoi(&*argv[1])-1;alg=atoi(&*argv[2]);
  }

    PoissonMatrix A(arg);
    LowerMatrix L(arg);
    UpperMatrix U(arg);
    Boundary B(arg,1);
    Startvector X(arg,0.0);
    Algorithms Run(arg);

```

```
24      double timer,start=0.0,end=0.0;
25      start=clock();
26
27      if(alg==1) {
28          method="Conjugate␣Gradient";
29          if(arg<512) steps=Run.CG(A,X.x,B.b,B.solved);
30          else steps=-1;
31      }
32      if(alg==2) {
33          method="P.␣Conjugate␣Gradient␣(ICG)";
34          if(arg<1024) {
35              A.InitHashMatrix();
36              Run.incompleteLU(A,L,U);
37              steps=Run.PCG(A,L,U,X.x,B.b,B.solved);
38          } else steps=-1;
39      }
40      if(alg==3) {
41          method="P.␣Conjugate␣Gradient␣(MICG)";
42          if(arg<1024) {
43              A.InitHashMatrix();
44              Run.modifiedIncompleteCholesky(A,L,U);
45              steps=Run.PCG(A,L,U,X.x,B.b,B.solved);
46          } else steps=-1;
47      }
48      if(alg==4) {
49          method="Two␣Grid";
50          if(arg<4096) steps=Run.MultiGridMethod(A,X.x,B.b,B.solved,alg);
51          else steps=-1;
52      }
53      if(alg==5) {
54          method="V-Cycle";
55          if(arg<4096) steps=Run.MultiGridMethod(A,X.x,B.b,B.solved,alg);
56          else steps=-1;
57      }
58      if(alg==6) {
59          method="W-Cycle";
60          if(arg<4096) steps=Run.MultiGridMethod(A,X.x,B.b,B.solved,alg);
61          else steps=-1;
62      }
63      if(alg==7) {
64          method="Jacobi␣Method";
65          if(arg<256) steps=Run.JacobiMethod(A,X.x,B.b,B.solved);
```

```
66          else steps=-1;
67      }
68      if(alg==8) {
69          method="Jacobi Relaxation Method";
70          if(arg<256) steps=Run.JacobiRelaxationMethod(A,X.x,B.b,B.solved)
                ;
71          else steps=-1;
72      }
73      if(alg==9) {
74          method="Gauss-Seidel-Method";
75          if(arg<256) steps=Run.GaussSeidelMethod(A,X.x,B.b,B.solved);
76          else steps=-1;
77      }
78      if(alg==10) {
79          method="SOR Method";
80          if(arg<512) steps=Run.SORMethod(A,X.x,B.b,B.solved);
81          else steps=-1;
82      }
83
84      end=clock();
85      timer=(end-start)/CLOCKS_PER_SEC;
86      printf("%s with %d steps in %f seconds.\n", method,steps,timer);
87      X.WriteToFile();
88
89      return EXIT_SUCCESS;
90  }
91
92  vector<double> operator-(const vector<double>& lhs, const vector<double
        >& rhs) {
93      vector<double> tmp(lhs);
94      for(int i=0;i<(int)lhs.size();i++) {
95          tmp[i]=lhs[i]-rhs[i];
96      }
97      return tmp;
98  }
99
100 void operator+=(vector<double>& lhs, const vector<double>& rhs) {
101     for(int i=0;i<(int)rhs.size();i++) {
102         lhs[i]+=rhs[i];
103     }
104 }
105
```

```
106 vector<double> operator*(double x, vector<double> rhs) {
107     vector<double> tmp(rhs);
108     for(int i=0;i<(int)rhs.size();i++) {
109         tmp[i]=x*rhs[i];
110     }
111     return tmp;
112 }
113
114 double operator|(const std::vector<double>& x,const std::vector<double>&
        y) {
115     double norm=0.0;
116     for(int i=0;i<(int)x.size();i++) {
117         norm+=x[i]*y[i];
118     }
119     return sqrt(norm);
120 }
121
122 double operator*(const std::vector<double>& x,const std::vector<double>&
        y) {
123     double ip=0.0;
124     for (int i=0;i<(int)x.size();i++)
125         ip+=x[i]*y[i];
126     return ip;
127 }
```

7.2 Klassen

```
1  #ifndef CLASSES_H
2  #define CLASSES_H
3
4  #include <iostream>
5  #include <cstdlib>
6  #include <cmath>
7  #include <fstream>
8  #include <cstdio>
9  #include <vector>
10 #include <assert.h>
11 #include <time.h>
12 using namespace std;
13
14 vector<double> operator-(const vector<double>&,const vector<double>&);
```

```
15 void operator+=(vector<double>&,const vector<double>&);
16 vector<double> operator*(double,vector<double>);
17 double operator|(const vector<double>&,const vector<double>&);
18 double operator*(const vector<double>&,const vector<double>&);
19
20 class Matrix {
21     public:
22         vector<vector<int> > HashMatrix;
23         virtual int Size()=0;
24         virtual double Get(int,int)=0;
25         virtual vector<double> operator*(const vector<double>&)=0;
26 };
27
28 class WriteableMatrix : public Matrix {
29     public:
30         virtual void Set(int,int,double)=0;
31 };
32
33 class PoissonMatrix : public Matrix {
34     private:
35         int dim;
36         int n;
37         double diagonal;
38         double tridiagonal;
39         double identity;
40     public:
41         PoissonMatrix(int);
42         ~PoissonMatrix();
43         int Size();
44         double Get(int, int);
45         void InitHashMatrix();
46         vector<double> operator*(const vector<double>&);
47 };
48
49 class LowerMatrix : public WriteableMatrix {
50     private:
51         int dim;
52         int n;
53     public:
54         LowerMatrix(int);
55         ~LowerMatrix();
56         vector<double> diagonal;
```

```
57          vector<double> tridiagonal;
58          vector<double> identity;
59          int Size();
60          double Get(int,int);
61          void Set(int,int,double);
62          vector<double> operator*(const vector<double>&);
63  };
64
65  class UpperMatrix : public LowerMatrix {
66     private:
67        int dim;
68        int n;
69     public:
70        UpperMatrix(int);
71        ~UpperMatrix();
72          double Get(int,int);
73        void Set(int,int,double);
74          vector<double> operator*(const vector<double>&);
75  };
76
77  class Vectors {
78      public:
79          virtual double Get(int)=0;
80          virtual int Size()=0;
81          void WriteToFile();
82          double f(double,double,int);
83          double g(double,double,int);
84  };
85
86  class Boundary : public Vectors {
87      private:
88          int dim;
89          int n;
90          double h;
91          int k;
92      public:
93          Boundary(int,int);
94          ~Boundary();
95          vector<double> b;
96          vector<double> solved;
97          double Get(int);
98          int Size();
```

```
99  };
100
101 class Startvector : public Vectors {
102     private:
103         int dim;
104         int n;
105         double value;
106     public:
107         Startvector(int,double);
108         ~Startvector();
109         vector<double> x;
110         double Get(int);
111         int Size();
112 };
113
114 class Algorithms {
115    private:
116         int n;
117         int dim;
118         double h;
119         int Vcounter;
120    public:
121         Algorithms(int);
122         ~Algorithms();
123         int JacobiMethod(Matrix&,vector<double>&,const vector<double>&,
                const vector<double>&);
124         int JacobiRelaxationMethod(Matrix&,vector<double>&,const vector<
                double>&,const vector<double>&);
125         int GaussSeidelMethod(Matrix&,vector<double>&,const vector<
                double>&,const vector<double>&);
126         int SORMethod(Matrix&,vector<double>&,const vector<double>&,
                const vector<double>&);
127         int CG(Matrix&,vector<double>&,const vector<double>&,const
                vector<double>&);
128         int PCG(Matrix&,WriteableMatrix&,WriteableMatrix&,vector<double
                >&,const vector<double>&,const vector<double>&);
129         void modifiedIncompleteCholesky(Matrix&, WriteableMatrix&,
                WriteableMatrix&);
130         void incompleteLU(Matrix&,WriteableMatrix&,WriteableMatrix&);
131         void LUsolverUpper(Matrix&,Matrix&,vector<double>&);
132
```

```
133         void JacobiRelaxation(Matrix&,vector<double>&,const vector<double>&,int);
134         void Restriction(const vector<double>&,vector<double>&,int);
135         void Interpolation(const vector<double>&,vector<double>&,int);
136         vector<double> Cycle(Matrix&,vector<double>&,const vector<double>&,int,int,Matrix&,WriteableMatrix&,WriteableMatrix&);
137         int MultiGridMethod(Matrix&,vector<double>&,const vector<double>&,const vector<double>&,int);
138         void PCGdirect(Matrix&,WriteableMatrix&,WriteableMatrix&,vector<double>&,const vector<double>&);
139         void CGdirect(Matrix&,vector<double>&,const vector<double>&);
140 };
141
142 #endif
```

7.3 Poisson Matrix Klassen

```
1  #include "classes.h"
2
3  PoissonMatrix::PoissonMatrix(int n) {
4     this->n=n;
5     this->dim=n*n;
6     this->diagonal=4.0*pow((n+1),2);
7     this->tridiagonal=-1.0*pow((n+1),2);
8     this->identity=-1.0*pow((n+1),2);
9  }
10
11 void PoissonMatrix::InitHashMatrix() {
12      vector<int> tmp(5,-1);
13      for(int i=0;i<this->dim;i++) {
14          if(i<this->n) {
15              if(i==0) {
16                  tmp[0]=i;
17                  tmp[1]=i+1;
18                  tmp[2]=i+this->n;
19                  tmp[3]=-1;
20                  tmp[4]=-1;
21              } else if(i%this->n!=this->n-1 && i!=0) {
22                  tmp[0]=i-1;
23                  tmp[1]=i;
24                  tmp[2]=i+1;
```

```
25                  tmp[3]=i+this->n;
26                  tmp[4]=-1;
27              } else {
28                  tmp[0]=i-1;
29                  tmp[1]=i;
30                  tmp[2]=i+this->n;
31                  tmp[3]=-1;
32                  tmp[4]=-1;
33              }
34          } else if(i>=this->n && i<this->dim-this->n) {
35              if(i%this->n==0) {
36                  tmp[0]=i-this->n;
37                  tmp[1]=i;
38                  tmp[2]=i+1;
39                  tmp[3]=i+this->n;
40                  tmp[4]=-1;
41              } else if(i%this->n==this->n-1) {
42                  tmp[0]=i-this->n;
43                  tmp[1]=i-1;
44                  tmp[2]=i;
45                  tmp[3]=i+this->n;
46                  tmp[4]=-1;
47              } else {
48                  tmp[0]=i-this->n;
49                  tmp[1]=i-1;
50                  tmp[2]=i;
51                  tmp[3]=i+1;
52                  tmp[4]=i+this->n;
53              }
54          } else if(i>=this->dim-this->n) {
55              if(i==this->dim-1) {
56                  tmp[0]=i-this->n;
57                  tmp[1]=i-1;
58                  tmp[2]=i;
59                  tmp[3]=-1;
60                  tmp[4]=-1;
61              } else if(i%this->n!=0 && i!=this->dim-1) {
62                  tmp[0]=i-this->n;
63                  tmp[1]=i-1;
64                  tmp[2]=i;
65                  tmp[3]=i+1;
66                  tmp[4]=-1;
```

```
67              } else {
68                  tmp[0]=i-this->n;
69                  tmp[1]=i;
70                  tmp[2]=i+1;
71                  tmp[3]=-1;
72                  tmp[4]=-1;
73              }
74          }
75          HashMatrix.push_back(tmp);
76      }
77  }
78
79  PoissonMatrix::~PoissonMatrix() {
80      vector<vector<int> >().swap(HashMatrix);
81  }
82
83  int PoissonMatrix::Size() {
84     return dim;
85  }
86
87  double PoissonMatrix::Get(int i,int j) {
88     if(i==j) {
89        return diagonal;
90     } else if((j==(i+1) && i%n!=(n-1)) || (j==(i-1) && i%n!=0)) {
91        return tridiagonal;
92     } else if((j==(i+n)) || (j==(i-n))) {
93        return identity;
94     } else {
95        return 0.0;
96     }
97  }
98
99  vector<double> PoissonMatrix::operator*(const vector<double>& x) {
100     int dim=x.size(),n=sqrt(dim);
101     vector<double> tmp(dim,0);
102     for(int i=0;i<dim;i++) {
103         tmp[i]+=x[i]*4.0*pow(n+1,2);
104         if(i<(dim-n)) {
105             tmp[i]+=x[i+n]*-1.0*pow(n+1,2);
106             tmp[i+n]+=x[i]*-1.0*pow(n+1,2);
107         }
108         if(i%n!=0) {
```

```
109             tmp[i]+=x[i-1]*-1.0*pow(n+1,2);
110             tmp[i-1]+=x[i]*-1.0*pow(n+1,2);
111         }
112     }
113     return tmp;
114 }
```

7.4 Cholesky-Zerlegung Klassen

```
1  #include "classes.h"
2
3  LowerMatrix::LowerMatrix(int n) {
4      this->n=n;
5      this->dim=n*n;
6      this->diagonal.assign(dim,1);
7      this->tridiagonal.assign(dim-1,0);
8      this->identity.assign(dim-n,0);
9  }
10
11 LowerMatrix::~LowerMatrix() {
12     vector<double>().swap(diagonal);
13     vector<double>().swap(tridiagonal);
14     vector<double>().swap(identity);
15 }
16
17 int LowerMatrix::Size() {
18     return dim;
19 }
20
21 void LowerMatrix::Set(int i,int j,double value) {
22     if(i==j) {
23         diagonal[i]=value;
24     } else if(j==(i-1)) {
25         tridiagonal[j]=value;
26     } else if(j==(i-n)) {
27         identity[j]=value;
28     }
29 }
30
31 double LowerMatrix::Get(int i,int j) {
32     if(i==j) {
```

```
33          return diagonal[i];
34      } else if(j==(i-1)) {
35          return tridiagonal[j];
36      } else if(j==(i-n)) {
37          return identity[j];
38      } else {
39          return 0.0;
40      }
41  }
42
43  std::vector<double> LowerMatrix::operator*(const std::vector<double>& z) {
44      dim=z.size();
45      std::vector<double> tmp(z);
46      for(int i=0;i<dim;i++) {
47          if(i>=n) {
48              tmp[i]-=tmp[i-n]*Get(i,i-n);
49          }
50          if(i%n!=0) {
51              tmp[i]-=tmp[i-1]*Get(i,i-1);
52          }
53          tmp[i]/=Get(i,i);
54      }
55      return tmp;
56  }
57
58  UpperMatrix::UpperMatrix(int n) : LowerMatrix(n) {
59     this->n=n;
60     this->dim=n*n;
61     this->diagonal.assign(dim,0);
62     this->tridiagonal.assign(dim-1,0);
63     this->identity.assign(dim-n,0);
64  }
65
66  UpperMatrix::~UpperMatrix() {
67     vector<double>().swap(diagonal);
68     vector<double>().swap(tridiagonal);
69     vector<double>().swap(identity);
70  }
71
72  double UpperMatrix::Get(int i,int j) {
73     return LowerMatrix::Get(j,i);
```

```
74 }
75
76 void UpperMatrix::Set(int i,int j,double value) {
77    return LowerMatrix::Set(j,i,value);
78 }
79
80 std::vector<double> UpperMatrix::operator*(const std::vector<double>& z) ↩
       {
81     std::vector<double> tmp(z);
82     for(int i=dim-1;i>=0;i--) {
83         tmp[i]-=tmp[i]*Get(i,i);
84         if(i<=(dim-n)) {
85             tmp[i]-=tmp[i+n]*Get(i,i+n);
86         }
87         if(i%n!=0) {
88             tmp[i]-=tmp[i+1]*Get(i,i+1);
89         }
90         tmp[i]/=Get(i,i);
91     }
92     return tmp;
93 }
```

7.5 Vektor Klassen

```
1  #include "classes.h"
2
3  double Vectors::f(double x,double y,int k) {
4      double val;
5      if(k==1) val=-4.0;
6      return val;
7  }
8
9  double Vectors::g(double x,double y,int k) {
10     double val;
11     if(k==1) val=pow(x,2)+pow(y,2);
12     return val;
13 }
14
15 void Vectors::WriteToFile() {
16     FILE *file;
17     file=fopen("../plot/plot.dat","w");
```

```
18      if(file==NULL)
19          printf("<td colspan=2>ERROR: Could not open file!</td>");
20      else {
21          for(int i=0,k=0;i<=sqrt(Size());i++) {
22              for(int j=0;j<=sqrt(Size());j++) {
23                  if(i==0) {
24                      continue;
25                  } else if(i!=0) {
26                      if(j==0) {
27                          continue;
28                      } else if(j!=0){
29                          fprintf(file,"%f %f %f\n",(double)j/(double)(
                                sqrt(Size())),(double)i/(double)(sqrt(Size())
                                ),Get(k));
30                          k++;
31                      }
32                  }
33              }
34              fprintf(file,"\n");
35          }
36      }
37      fclose (file);
38  }
39
40  Boundary::Boundary(int n,int k) {
41      this->n=n;
42      this->dim=n*n;
43      this->h=1.0/(double)(n+1);
44      this->k=k;
45      this->b.resize(dim);
46      this->solved.resize(dim);
47
48      for(int i=1,k=0;i<=n;i++) {
49          for(int j=1;j<=n;j++,k++) {
50              solved[k]=g(j*h,i*h,this->k);
51              b[k]=f(i*h,j*h,this->k);
52              if(i==1) b[k]+=pow(1.0/h,2)*g(j*h,0,this->k);
53              if(i==n) b[k]+=pow(1.0/h,2)*g(j*h,1,this->k);
54              if(j==1) b[k]+=pow(1.0/h,2)*g(0,i*h,this->k);
55              if(j==n) b[k]+=pow(1.0/h,2)*g(1,i*h,this->k);
56          }
57      }
```

```
58 }
59
60 Boundary::~Boundary() {
61     vector<double>().swap(b);
62 }
63
64 double Boundary::Get(int i) {
65     return this->b[i];
66 }
67
68 int Boundary::Size() {
69     return b.size();
70 }
71
72 Startvector::Startvector(int n,double value) {
73     this->n=n;
74     this->dim=n*n;
75     this->value=value;
76     this->x.assign(dim,value);
77 }
78
79 Startvector::~Startvector() {
80     vector<double>().swap(x);
81 }
82
83 double Startvector::Get(int i) {
84     return this->x[i];
85 }
86
87 int Startvector::Size() {
88     return x.size();
89 }
```

7.6 Algorithmen

```
1 #include "classes.h"
2
3 Algorithms::Algorithms(int n) {
4     this->n=n;
5     this->dim=n*n;
6     this->h=1.0/(double)(n+1);
```

```
7       this->Vcounter=0;
8   }
9
10  Algorithms::~Algorithms() {}
11
12  void Algorithms::PCGdirect(Matrix& A,WriteableMatrix& L,WriteableMatrix&
        U,vector<double>& x,const vector<double>& b) {
13      int dim=x.size();
14      double alpha,beta=0.0,num1,num2,denom;
15
16      vector<double> r(dim),Ap(dim);
17      r=b-A*x;
18      vector<double> z(r),rTmp(r);
19      z=L*z;
20      LUsolverUpper(A,U,z);
21      vector<double> p(z),zTmp(z);
22
23      num2=zTmp*rTmp;
24      num1=num2;
25      double TOL=pow(10,-2)*(r|r);
26      while(TOL<(r|r)) {
27          Ap=A*p;
28          denom=p*Ap;
29          alpha=num2/denom;
30
31          for(int i=0;i<dim;i++) {
32              x[i]+=alpha*p[i];
33              r[i]-=alpha*Ap[i];
34          }
35
36          z=r;
37          z=L*z;
38          LUsolverUpper(A,U,z);
39          zTmp=z;
40
41          num2=z*r;
42          beta=num2/num1;
43
44          for(int i=0;i<dim;i++) {
45              p[i]=z[i]+beta*p[i];
46          }
47          rTmp=r;
```

```
48          num1=num2;
49      }
50  }
51
52  void Algorithms::CGdirect(Matrix& A,vector<double>& x,const vector<
        double>& b) {
53      double alpha,beta=0.0,num1,num2,denom;
54      int dim=x.size();
55
56      vector<double> r(dim),Ap(dim);
57      r=b-A*x;
58      vector<double> p(r),rTmp(r);
59
60      num2=rTmp*rTmp;
61      num1=num2;
62      double TOL=pow(10,-3)*(r|r);
63      while(TOL<(r|r)) {
64          Ap=A*p;
65          denom=p*Ap;
66          alpha=num2/denom;
67
68          for(int i=0;i<dim;i++) {
69              x[i]+=alpha*p[i];
70              r[i]-=alpha*Ap[i];
71          }
72
73          num2=r*r;
74          beta=num2/num1;
75
76          for(int i=0;i<dim;i++) {
77              p[i]=r[i]+beta*p[i];
78          }
79          rTmp=r;
80          num1=num2;
81      }
82  }
83
84  void Algorithms::JacobiRelaxation(Matrix& A, vector<double>& x, const
        vector<double>& b, int steps) {
85      vector<double> tmp;
86      tmp=x;
87      double sum,omega=4.0/5.0;
```

```
88      int n=sqrt(x.size()),dim=n*n;
89      for(int k=0;k<steps;k++) {
90          for(int i=0;i<n*n;i++) {
91              sum=0.0;
92              if(i>=n && i<dim-n) sum+=tmp[i-n]+tmp[i+n];
93              if(i<n) sum+=tmp[i+n];
94              if(i>=dim-n) sum+=tmp[i-n];
95              if(i%n!=0) sum+=tmp[i-1];
96              if(i%n!=n-1) sum+=tmp[i+1];
97              x[i]=omega*1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum)+
                    tmp[i]*(1-omega);
98              tmp[i]=x[i];
99          }
100     }
101 }
102
103 void Algorithms::Restriction(const vector<double>& r,vector<double>& r2h
        ,int n) {
104     for(int i=1,l=0,k=0;i<=n;i++) {
105         for(int j=1;j<=n;j++,k++) {
106             if(i%2==0 && j%2==0) {
107                 r2h[l]=1.0/16.0*(4.0*r[k]+2.0*(r[k-1]+r[k+1]+r[k-n]+r[k+
                        n])+r[k+n-1]+r[k+n+1]+r[k-n-1]+r[k-n+1]);
108                 l++;
109             }
110         }
111     }
112 }
113
114 void Algorithms::Interpolation(const vector<double>& E2h,vector<double>&
        E,int n) {
115     for(int i=1,k=0,l=0;i<=n;i++) {
116         for(int j=1;j<=n;j++,k++) {
117             if(i%2==0 && j%2==0) {
118                 E[k]=E2h[l];
119                 l++;
120             }
121         }
122     }
123     for(int i=1,k=0;i<=n;i++) {
124         for(int j=1;j<=n;j++,k++) {
125             if(i%2==0 && j%2!=0) {
```

```
126                 if(j!=1 && j!=n) E[k]=1.0/2.0*(E[k-1]+E[k+1]);
127                 if(j==1) E[k]=1.0/1.0*(E[k+1]);
128                 if(j==n) E[k]=1.0/1.0*(E[k-1]);
129             }
130             if(i%2!=0 && j%2==0) {
131                 if(i!=1 && i!=n) E[k]=1.0/2.0*(E[k-n]+E[k+n]);
132                 if(i==1) E[k]=1.0/1.0*(E[k+n]);
133                 if(i==n) E[k]=1.0/1.0*(E[k-n]);
134             }
135             if(i%2!=0 && j%2!=0) {
136                 if(i!=1 && j!=1 && i!=n && j!=n) E[k]=1.0/4.0*(E[k+n-1]+
                        E[k+n+1]+E[k-n+1]+E[k-n-1]);
137                 if(i==1 && j==1) E[k]=1.0/1.0*(E[k+n+1]);
138                 if(i==n && j==n) E[k]=1.0/1.0*(E[k-n-1]);
139                 if(i==1 && j==n) E[k]=1.0/1.0*(E[k+n-1]);
140                 if(i==n && j==1) E[k]=1.0/1.0*(E[k-n+1]);
141                 if(i==1 && j!=1 && j!=n) E[k]=1.0/2.0*(E[k+n+1]+E[k+n
                        -1]);
142                 if(i==n && j!=1 && j!=n) E[k]=1.0/2.0*(E[k-n-1]+E[k-n
                        +1]);
143                 if(j==1 && i!=1 && i!=n) E[k]=1.0/2.0*(E[k+n+1]+E[k-n
                        +1]);
144                 if(j==n && i!=1 && i!=n) E[k]=1.0/2.0*(E[k+n-1]+E[k-n
                        -1]);
145             }
146
147         }
148     }
149 }
150
151 vector<double> Algorithms::Cycle(Matrix& A,vector<double>& x,const
        vector<double>& b,int lambda,int theta,Matrix& B,WriteableMatrix& L,
        WriteableMatrix& U) {
152     int dim=x.size(),n=sqrt(dim),N2h=(n+1)/2-1,dim2h=pow(N2h,2);
153     vector<double> r(dim,0),E(dim,0),r2h(dim2h,0),E2h(dim2h,0);
154     if(this->Vcounter==lambda) {
155         PCGdirect(B,L,U,x,b);
156         // CGdirect(A,x,b);
157         return x;
158     } else {
159         this->Vcounter++;
160         JacobiRelaxation(A,x,b,3);
```

```
161         r=b-A*x;
162         Restriction(r,r2h,n);
163         E2h=Cycle(A,E2h,r2h,lambda,theta,B,L,U);
164         if(theta==1) E2h=Cycle(A,E2h,r2h,lambda,theta,B,L,U);
165         Interpolation(E2h,E,n);
166         x+=E;
167         JacobiRelaxation(A,x,b,3);
168         this->Vcounter--;
169         return x;
170     }
171 }
172
173 int Algorithms::MultiGridMethod(Matrix& A,vector<double>& x,const vector
    <double>& b,const vector<double>& solved,int alg) {
174     int steps=0,dim=x.size();
175     vector<double> r(dim);
176     r=x-solved;
177     int numberOfGrids=2,VW=0,n=0;
178     if(alg==4) {
179         numberOfGrids=1;
180         VW=0;
181         n=((int)sqrt(dim)+1)/2-1;
182     }
183     if(alg==5) {
184         numberOfGrids=2;
185         VW=0;
186         n=((int)sqrt(dim)+1)/4-1;
187     }
188     if(alg==6) {
189         numberOfGrids=2;
190         VW=1;
191         n=((int)sqrt(dim)+1)/4-1;
192     }
193     PoissonMatrix B(n);
194     LowerMatrix L(n);
195     UpperMatrix U(n);
196     B.InitHashMatrix();
197     modifiedIncompleteCholesky(B,L,U);
198     double TOL=pow(10,-3)*(r|r);
199     while(TOL<=(r|r)) {
200         x=Cycle(A,x,b,numberOfGrids,VW,B,L,U);
201         r=x-solved;
```

```
202         steps++;
203     }
204     return steps;
205 }
206
207 int Algorithms::JacobiMethod(Matrix& A,vector<double>& x,const vector<double>& b,const vector<double>& solved) {
208     vector<double> tmp;
209     double sum;
210     int n=sqrt(x.size()),dim=n*n,steps=0;
211     double h=1.0/(double)(n+1);
212     vector<double> r(x.size());
213     r=x-solved;
214     double TOL=(r|r)*pow(10,-3);
215     while(TOL<=(r|r)) {
216         tmp=x;
217         for(int i=0;i<n*n;i++) {
218             sum=0.0;
219             if(i>=n && i<dim-n) sum+=tmp[i-n]+tmp[i+n];
220             if(i<n) sum+=tmp[i+n];
221             if(i>=dim-n) sum+=tmp[i-n];
222             if(i%n!=0) sum+=tmp[i-1];
223             if(i%n!=n-1) sum+=tmp[i+1];
224             x[i]=1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum);
225             r[i]=x[i]-solved[i];
226         }
227         steps++;
228     }
229     return steps;
230 }
231
232 int Algorithms::JacobiRelaxationMethod(Matrix& A,vector<double>& x,const vector<double>& b,const vector<double>& solved) {
233     vector<double> tmp;
234     double sum,omega=4.0/5.0;
235     int n=sqrt(x.size()),dim=n*n,steps=0;
236     double h=1.0/(double)(n+1);
237     vector<double> r(x.size());
238     r=x-solved;
239     double TOL=(r|r)*pow(10,-3);
240     while(TOL<=(r|r)) {
241         tmp=x;
```

```
242         for(int i=0;i<n*n;i++) {
243             sum=0.0;
244             if(i>=n && i<dim-n) sum+=tmp[i-n]+tmp[i+n];
245             if(i<n) sum+=tmp[i+n];
246             if(i>=dim-n) sum+=tmp[i-n];
247             if(i%n!=0) sum+=tmp[i-1];
248             if(i%n!=n-1) sum+=tmp[i+1];
249             x[i]=omega*1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum)+
                    tmp[i]*(1-omega);
250             r[i]=x[i]-solved[i];
251         }
252         steps++;
253     }
254     return steps;
255 }
256
257 int Algorithms::GaussSeidelMethod(Matrix& A,vector<double>& x,const
    vector<double>& b,const vector<double>& solved) {
258     double sum;
259     int n=sqrt(x.size()),dim=n*n,steps=0;
260     double h=1.0/(double)(n+1);
261     vector<double> r(x.size());
262     r=x-solved;
263     double TOL=(r|r)*pow(10,-3);
264     while(TOL<=(r|r)) {
265         for(int i=0;i<n*n;i++) {
266             sum=0.0;
267             if(i>=n && i<dim-n) sum+=x[i-n]+x[i+n];
268             if(i<n) sum+=x[i+n];
269             if(i>=dim-n) sum+=x[i-n];
270             if(i%n!=0) sum+=x[i-1];
271             if(i%n!=n-1) sum+=x[i+1];
272             x[i]=1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum);
273             r[i]=x[i]-solved[i];
274         }
275         steps++;
276     }
277     return steps;
278 }
279
280 int Algorithms::SORMethod(Matrix& A,vector<double>& x,const vector<
    double>& b,const vector<double>& solved) {
```

```
281     double sum,Pi=3.141592654,omega=2/(1+sqrt(1-pow(cos(Pi*h),2)));
282     int n=sqrt(x.size()),dim=n*n,steps=0;
283     double h=1.0/(double)(n+1);
284     vector<double> r(x.size());
285     r=x-solved;
286     double TOL=(r|r)*pow(10,-3);
287     while(TOL<=(r|r)) {
288         for(int i=0;i<n*n;i++) {
289             sum=0.0;
290             if(i>=n && i<dim-n) sum+=x[i-n]+x[i+n];
291             if(i<n) sum+=x[i+n];
292             if(i>=dim-n) sum+=x[i-n];
293             if(i%n!=0) sum+=x[i-1];
294             if(i%n!=n-1) sum+=x[i+1];
295             x[i]=omega*1.0/(4.0*pow(1.0/h,2))*(b[i]+pow(1.0/h,2)*sum)+x[
                    i]*(1-omega);
296             r[i]=x[i]-solved[i];
297         }
298         steps++;
299     }
300     return steps;
301 }
302
303 int Algorithms::CG(Matrix& A, vector<double>& x,const vector<double>& b,
        const vector<double>& solved) {
304     double alpha,beta=0.0,num1,num2,denom;
305     int steps=0;
306
307     vector<double> r(dim),Ap(dim),tmp(dim,1);
308     r=b-A*x;
309     vector<double> p(r),rTmp(r);
310
311     num2=rTmp*rTmp;
312     num1=num2;
313     double TOL=pow(10,-3)*((x-solved)|(x-solved));
314     while(TOL<(tmp|tmp)) {
315         Ap=A*p;
316         denom=p*Ap;
317         alpha=num2/denom;
318
319         for(int i=0;i<dim;i++) {
320             x[i]+=alpha*p[i];
```

```
321             r[i]-=alpha*Ap[i];
322         }
323
324         num2=r*r;
325         beta=num2/num1;
326
327         for(int i=0;i<dim;i++) {
328             p[i]=r[i]+beta*p[i];
329             tmp[i]=x[i]-solved[i];
330         }
331         rTmp=r;
332         num1=num2;
333         steps++;
334     }
335     return steps;
336 }
337
338 int Algorithms::PCG(Matrix& A,WriteableMatrix& L,WriteableMatrix& U,
    vector<double>& x,const vector<double>& b,const vector<double>&
    solved) {
339     double alpha,beta=0.0,num1,num2,denom;
340     int steps=0;
341
342     vector<double> r(dim),Ap(dim),tmp(dim,1);
343     r=b-A*x;
344     vector<double> z(r),rTmp(r);
345     z=L*z;
346     LUsolverUpper(A,U,z);
347     vector<double> p(z),zTmp(z);
348
349     num2=zTmp*rTmp;
350     num1=num2;
351     double TOL=pow(10,-3)*((x-solved)|(x-solved));
352     while(TOL<(tmp|tmp)) {
353         Ap=A*p;
354         denom=p*Ap;
355         alpha=num2/denom;
356
357         for(int i=0;i<dim;i++) {
358             x[i]+=alpha*p[i];
359             r[i]-=alpha*Ap[i];
360         }
```

```
361
362         z=r;
363         z=L*z;
364         LUsolverUpper(A,U,z);
365         zTmp=z;
366
367         num2=z*r;
368         beta=num2/num1;
369
370         for(int i=0;i<dim;i++) {
371             p[i]=z[i]+beta*p[i];
372             tmp[i]=x[i]-solved[i];
373         }
374         rTmp=r;
375         num1=num2;
376         steps++;
377     }
378     return steps;
379 }
380
381 void Algorithms::modifiedIncompleteCholesky(Matrix& A,WriteableMatrix& L
    ,WriteableMatrix& U) {
382     int i,j,k,m,u,dim=A.Size();
383     double sum, drop;
384
385     for(i=0;i<dim;i++) {
386         drop=0;
387         for(k=0;k<5;k++) {
388             m=A.HashMatrix[i][k];
389             if(m!=-1 && m<i) {
390                 sum=0;
391                 for(j=0;j<5;j++) {
392                     u=A.HashMatrix[i][j];
393                     if(u!=-1 && u<k) {
394                         sum+=L.Get(i,u)*U.Get(u,m);
395                     }
396                 }
397                 L.Set(i,m,(A.Get(i,m)-sum)/U.Get(m,m));
398                 drop+=sum;
399             } else if(m!=-1 && m>=i) {
400                 m=A.HashMatrix[i][k];
401                 if(m!=-1 && m>=i) {
```

```
402                     sum=0;
403                     for(j=0;j<5;j++) {
404                         u=A.HashMatrix[i][j];
405                         if(u!=-1 && u<i) {
406                             sum+=L.Get(i,u)*U.Get(u,m);
407                         }
408                     }
409                     U.Set(i,m,(A.Get(i,m)-sum));
410                     drop+=sum;
411                 }
412             }
413         }
414         U.Set(i,i,(U.Get(i,i)-drop));
415     }
416 }
417
418 void Algorithms::incompleteLU(Matrix& A, WriteableMatrix& L, ↩
    WriteableMatrix& U) {
419     int m,u;
420     double sum;
421
422     for(int i=0;i<dim;i++) {
423         for(int k=0;k<5;k++) {
424             m=A.HashMatrix[i][k];
425             if(m!=-1 && m<i) {
426                 sum=0;
427                 for(int j=0;j<5;j++) {
428                     u=A.HashMatrix[i][j];
429                     if(u!=-1 && u<k) {
430                         sum+=L.Get(i,u)*U.Get(u,m);
431                     }
432                 }
433                 L.Set(i,m,((A.Get(i,m)-sum)/U.Get(m,m)));
434             } else if(m!=-1 && m>=i) {
435                 sum=0;
436                 for(int j=0;j<5;j++) {
437                     u=A.HashMatrix[i][j];
438                     if(u!=-1 && u<i) {
439                         sum+=L.Get(i,u)*U.Get(u,m);
440                     }
441                 }
442                 U.Set(i,m,(A.Get(i,m)-sum));
```

```
443             }
444         }
445     }
446 }
447
448 void Algorithms::LUsolverUpper(Matrix& A,Matrix& U,vector<double>& z) {
449     int m;
450     int dim=z.size();
451     for(int i=dim-1;i>=0;i--) {
452         for(int j=0;j<5;j++) {
453             m=A.HashMatrix[i][j];
454             if(m!=-1 && m>=i) {
455                 z[i]-=U.Get(i,m)*z[m];
456             }
457          }
458         z[i]/=U.Get(i,i);
459     }
460 }
```

7.7 Makefile

```
1  CC = g++ -Wall -g
2
3  poissonSolver: PoissonMatrix.o Algorithms.o Vector.o LUMatrices.o
       PoissonSolver.o
4    $(CC) -o $@ $+
5
6  PoissonMatrix.o: PoissonMatrix.cpp classes.h
7    $(CC) -c -o $@ $<
8
9  Algorithms.o: Algorithms.cpp classes.h
10   $(CC) -c -o $@ $<
11
12 Vector.o: Vector.cpp classes.h
13   $(CC) -c -o $@ $<
14
15 LUMatrices.o: LUMatrices.cpp classes.h
16   $(CC) -c -o $@ $<
17
18 PoissonSolver.o: PoissonSolver.cpp classes.h
19   $(CC) -c -o $@ $<
```

```
20
21 clean:
22   rm *.o
```

7.8 Gnuplot Datei

```
1 set title "Plot␣function"
2 splot "../plot/plot.dat" t "3D␣Plot" with pm3d
```

Danksagung

An dieser Stelle möchte ich mich noch bei einigen Menschen bedanken, die mich bei der Erstellung dieser Bachelorarbeit unterstützt und ausgehalten haben:

- *Meiner Mama*: Für die Unterstützung, das Vertrauen in mich und die große Geduld mit mir.
- *Meiner Freundin*: Ohne dich, Sylvia, wäre ich nie durch dieses Studium gekommen. Deine Hilfe, dein Verständnis und dein Zuspruch haben mir sehr geholfen.
- *Meinen besten Freunden*: Obwohl ihr Mathematik alle nicht sonderlich mögt, habt ihr euch meine scheinbar endlosen Vorträge immer wieder angehört.
- *Christopher Rupprecht*: Die gute Hilfestellung hat mir gerade bei meinen Algorithmen sehr weiter geholfen.
- *Prof. Dr. Harald Garcke*: Vielen Dank für die kompetente Unterstützung und die investierte Zeit. Es ist nicht selbstverständlich, dass ein Professor immer zur Verfügung steht und ein offenes Ohr für seine Studenten hat. Durch ihre Einbringung und Hilfe war es mir möglich diese Arbeit auf ein gutes Niveau zu bringen. Vielen Dank für die hervorragende Betreuung!

Literaturverzeichnis

[ALO1] Götz Alefeld, Ingrid Lenhardt und Holger Obermaier, Parallele numerische Verfahren, Seiten 18-19, Springer-Verlag Berlin Heidelberg, 2002.

[ALO2] Götz Alefeld, Ingrid Lenhardt und Holger Obermaier, Parallele numerische Verfahren, Seite 21, Springer-Verlag Berlin Heidelberg, 2002.

[DR1] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 470-472, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR2] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seite 479, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR3] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seite 480, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR4] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 549-583, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR5] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 550, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR6] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 566, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR7] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 567, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR8] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 569, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[DR9] Wolfgang Dahmen und Arnold Reusken, Numerik für Ingeneure und Naturwissenschaftler, 2.,korrigierte Auflage, Seiten 570, Springer-Verlag Berlin Heidelberg, 2008, 2006.

[GL] G. H. Golub, C. F. van Loan, Matrix Computations, 3. Aufl., Oxford, University Press, 1996.

[SAAD1] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seite 411, SIAM Verlag, 2003.

[SAAD2] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 414-416, SIAM Verlag, 2003.

[SAA3] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 415, SIAM Verlag, 2003.

[SAAD4] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 416, SIAM Verlag, 2003.

[SAAD5] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 105-106, SIAM Verlag, 2003.

[SAAD6] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 407, SIAM Verlag, 2003.

[SAAD7] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 419-429, SIAM Verlag, 2003.

[SAAD8] Yousef Saad, Iterative Methods for sparse linear systems, 2nd ed., Seiten 435-437, SIAM Verlag, 2003.

[STR] Gilbert Strang, Wissenschaftliches Rechnen, ab Seiten 661-676, Springer-Verlag Berlin Heidelberg, 2010.