Bachelorarbeit

Die inkomplette LR-Zerlegung für Konvektions-Diffusionsgleichungen

Nils Christian Ehmke

Matr.-Nr 834645

Sommersemester 2011

17. September 2011

Betreuer: Prof. Dr. Malte Braack

Christian-Albrechts-Universität zu Kiel

Zusammenfassung

Diese Arbeit befasst sich mit den Konvektions-Diffusionsgleichungen, die hier kurz vorgestellt und dann mit der Finite-Differenzen-Methode diskretisiert werden. Mithilfe des stationären Richardson-Verfahrens als Lösungsverfahren und der unvollständigen LR-Zerlegung als Vorkonditionierer wird ein C++-Programm entwickelt, welches die für die diskretisierte Konvektions-Diffusionsgleichung notwendige Koeffizientenmatrix aufstellt, das resultierende lineare Gleichungssystem löst und das Ergebnis zweidimensional visuell darstellt. Weiterführende Vorkonditionierer und Verfahren zur Konvergenzbeschleunigung, sowie zur Stabilisierung werden vorgestellt, jedoch nicht tiefergehend behandelt.

Inhaltsverzeichnis

Nomenklatur

β	Richtung und Geschwindigkeit des Konvektionsterms
β_1	Erste Komponente des Vektors β
β_2	Zweite Komponente des Vektors β
$\text{cond}(A)$	Konditionszahl der Matrix A
Δ	Laplace-Operator
ϵ	Diffusionskoeffizient
$\mathcal{O}(\cdot)$	Landau-Symbol für die asymptotische obere Schranke
$\|\cdot\|_1$	l1-Norm
$\|\cdot\|_\infty$	Supremumsnorm
∇	Nabla-Operator
Ω	Umgebung
$\partial\Omega$	Rand der Umgebung Ω
e	Fehlerschranke
$E(A)$	Besetzungsstruktur der Matrix A
f	Quellterm
h	Gitterhöhe des Diskretisierungsgitters
L	Linke untere Dreiecksmatrix der Zerlegung
L_h	Systemmatrix der diskretisierten Poisson- bzw. Konvektions-Diffusionsgleichung
n	Gitteranzahl des Diskretisierungsgitters
U	Rechte obere Dreiecksmatrix der Zerlegung
x	Erste Richtung im kartesischen, zweidimensionalen Koordinatensystem
y	Zweite Richtung im kartesischen, zweidimensionalen Koordinatensystem
I	Einheitsmatrix

1. Einleitung

1.1. Vorwort

Die Numerik ist ein Teilgebiet der Mathematik, welche zunehmend in Bereiche der Physik und weitere naturwissenschaftliche Gebiete eindringt und für diese mehr und mehr unabdingbar wird. So existieren mittlerweile parametrisierte Simulationen, die so komplex und detailliert sind, dass sie ohne Hochleistungsrechner nicht mehr durchgeführt werden können. Beispielsweise werden in der Biologie die Vermehrung von Bakterien oder der Membrantransport, in der Meteorologie das Verhalten von Wetter und Klima und in der Physik die Strömungsmechanik mit entsprechenden Modellen numerisch simuliert. Neben Simulationen kommt die Numerik vor allem auch bei Problemen zur Anwendung, bei denen keine explizite Darstellung der Lösung existiert, die Lösung zu aufwändig zu berechnen ist oder analytische Lösungsverfahren schlicht an ihre Grenzen stoßen.

Gerade in der Physik führen zu simulierende Systeme mit dynamischen Eigenschaften sehr häufig auf gewöhnliche oder partielle Differentialgleichungen. Selbst einfachste Bewegungsgleichungen lassen sich, sobald die verschiedenen auf die Bewegung wirkenden Kräfte einbezogen werden, oft nur durch Differentialgleichungen beschreiben, deren Lösung nicht in einer geschlossenen Form berechnet werden kann. Die Diskretisierung dieser kontinuierlichen Systeme und das Finden effizienter numerischer Verfahren zur näherungsweisen Bestimmung der Lösungen ist somit unverzichtbar. Hierbei wird ein Augenmerk nicht nur auf schnelle Algorithmen gelegt, sondern vor allem auf solche, die mit möglichst wenig Speicherbedarf auskommen und gleichzeitig numerisch stabil sind. Algorithmen ohne diese Eigenschaften wären für die praktische Anwendung untauglich.

Diese Bachelorarbeit befasst sich im Speziellen mit Konvektions-Diffusionsgleichungen und - als Spezialfall - mit der Poisson-Gleichung. Konvektions-Diffusionsgleichungen sind Gleichungen, die häufig als Teilproblem bei der Simulation von Strömungen in der Physik auftauchen und entsprechend fundamental sind, um anhand dieser ein mögliches numerisches Verfahren zu demonstrieren.

In dieser Arbeit wird dazu zunächst darauf eingegangen, um was es sich bei diesen Konvektions-Diffusionsgleichungen handelt, ehe die Diskretisierung des Problems und die Herleitung des dazugehörigen linearen Gleichungssystems beschrieben wird. Unabhängig von dem spezifischen Problem wird mit dem stationären Richardson-Verfahren ein iteratives Lösungsverfahren für lineare Gleichungssysteme vorgestellt, welches mit der

inkompletten LR-Zerlegung als Vorkonditionierer kombiniert wird. Das Zusammenführen der Konvektions-Diffusionsgleichungen mit der modifizierten Richardson-Iteration wird dann im Rahmen dieser Bachelorarbeit in einem C++-Computerprogramm durchgeführt. Anhand dieses Programms wird dann die Ausführungsgeschwindigkeit bei verschiedenen Parameterkonstellationen gemessen. Anschließend wird in dieser Arbeit auf Möglichkeiten der Konvergenzbeschleunigung und numerischen Stabilisierung eingegangen.

Die Eigenarten der Konvektions-Diffusionsgleichungen, der Aufbau und die Eigenschaften der bei der Diskretisierung entstehenden Matrizen wurden ebenso wie die Algorithmen für diese Bachelorarbeit der Literatur entnommen, wobei ein erheblicher Teil auf den theoretischen Betrachtungen von W. Hackbusch ([Hac86] und [Hac93]) basiert. Die tatsächliche Implementierung und Analyse des Programms sowie die Auswertungen der entstandenen Messdaten wurde hingegen ohne die Inanspruchnahme weiterer Literatur vorgenommen.

1.2. Einführung in die Konvektions-Diffusionsgleichungen

Die nach dem französischen Mathematiker und Physiker Siméon-Denis Poisson [Enc11] benannte Poisson-Gleichung [Hac86] ist eine elliptische partielle Differentialgleichung, die vor allem in der Physik eine grundlegende Rolle spielt und häufig als Komponente von komplexeren Randwertproblemen Anwendung findet. Es handelt sich um ein Modellproblem, welches als einfachstes, nichttriviale Beispiel einer Randwertaufgabe gilt (Vgl. [Hac93]). So lässt sich mit ihr etwa der Zusammenhang zwischen der Massedichte und dem Gravitationspotential in der Gravitationslehre oder der Zusammenhang zwischen der Ladungsdichte und dem elektrischen Potential in der Elektrodynamik beschreiben. Die Poisson-Gleichung, wie sie in dieser Arbeit verwendet wird, lässt sich Formel (1.1) entnehmen.

$$-\Delta u = f \text{ in } \Omega \tag{1.1}$$

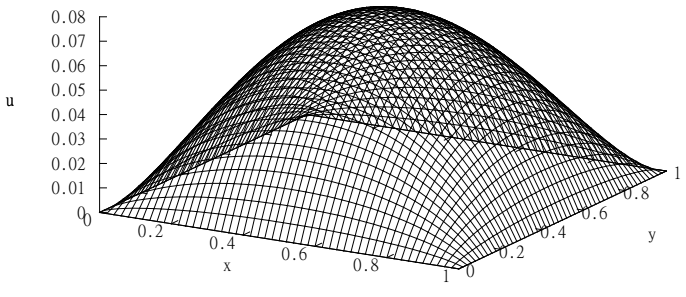


Abbildung 1.1.: Dreidimensionale Visualisierung der diskretisierten Poisson-Gleichung ($n = 50$, $e = 10^{-4}$, $f \equiv 1$)

Hierbei entspricht $f: \Omega \to \mathbb{R}$ einer vorgegebenen Funktion (dem sogenannten "Quellterm") und $\Omega \subset \mathbb{R}^m$ einer Umgebung (in der Physik auch als "Gebiet" bezeichnet). Der homogene Spezialfall $f \equiv 0$ wird als Laplace-Gleichung bezeichnet [Hac86]. Für die gesuchte Lösung $u: \Omega \to \mathbb{R}$ wird eine Funktion vorausgesetzt, welche im Gebiet zweimal stetig differenzierbar sein muss, das heißt $u \in C^2(\Omega) \cap C^0(\partial\Omega)$. Um die Lösung eindeutig bestimmen zu können, müssen weitere Daten gegeben sein, die das Verhalten der Funktion auf dem Rand beschreiben (Randbedingungen). Übliche Formen sind

- **Dirichlet-Randbedingungen**
  Diese Bedingungen schreiben vor, wie sich die Funktion u exakt auf dem Rand verhält und haben die Form

$$u(x, y) = \varphi(x, y) \text{ auf } \partial\Omega. \tag{1.2}$$

und

- **Neumann-Randbedingungen**
  Die Neumann-Randbedingungen beschreiben anschaulich "wieviel in Normalenrichtung ins Gebiet hineinfließt" [EGK08] und haben die Form

$$\frac{\partial}{\partial n} u(x, y) = \varphi(x, y) \text{ auf } \partial\Omega. \tag{1.3}$$

In dieser Arbeit wird vereinfacht von den homogenen Dirichlet-Randbedingungen

$$u = 0 \text{ auf } \partial\Omega \tag{1.4}$$

ausgegangen (An dieser Stelle sei angemerkt, dass sich auch nicht-homogene Dirichlet-Randbedingungen auf homogene Bedingungen transformieren lassen [Bra92]). Weiterhin

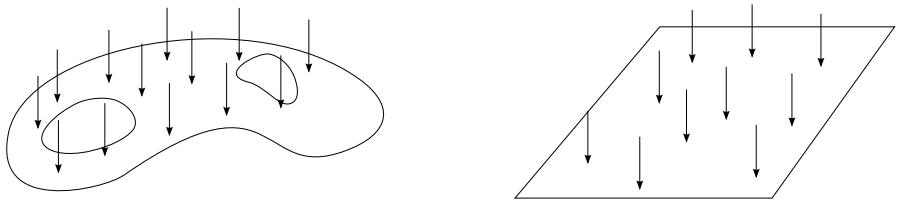
werden die hier aufgeführten Gleichungen auf dem Raum $\mathbb{R}^2$ betrachtet und Ω zudem aus Gründen der Diskretisierung als das Einheitsquadrat

$$\Omega = (0,1)^2 \tag{1.5}$$

gewählt. Unter der Voraussetzung der Wahl einer passenden Gitterweite des Diskretisierungsgitters ließen sich jedoch auch beispielsweise beliebige Rechtecke als Umgebung nutzen. Mit der Anwendung des Laplace-Operators [Hac93] lässt sich die Poisson-Gleichung so wie in Formel (1.6) angegeben darstellen.

$$\begin{aligned} -\Delta u = -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) &= f \text{ in } \Omega \\ u &= 0 \text{ auf } \partial\Omega \end{aligned} \tag{1.6}$$

Als beispielhafte Darstellung der Poisson-Gleichung im Raum $\mathbb{R}^2$ mit homogenen Dirichlet-Randbedingungen kann man sich eine Membran (Umgebung) vorstellen, die an den Seiten fest eingespannt wird (Randbedingungen) und auf die eine gewisse Kraft (gegeben durch die Funktion f) einwirkt. Die Suche nach der Lösung u entspricht nun der Suche nach einer Funktion, welche die Spannung an jedem Punkt der Umgebung beschreibt. Dies ist auch in Abbildung 1.2 noch einmal dargestellt. Speziell auf die Strömungsmechanik bezogen, beschreibt die Poisson-Gleichung die Verteilung einer physikalischen Größe u in einem stationären ruhenden Fluid (Vgl. [Bad01]).



(a) Beliebige Umgebung (b) Einheitsquadrat als Umgebung

Abbildung 1.2.: Membran auf welcher die Kraft wirkt

In der praktischen Anwendung passiert es häufig, dass das erwähnte Fluid zwar weiterhin zeitlich stationär bleibt, sich jedoch bewegt (Vgl. [Bad01]). Dann müssen der obigen Poisson-Gleichung noch zusätzliche Daten in Form von Advektion und Konvektion [Bad01] hinzugefügt werden, was dann als Advektions-Diffusionsgleichung oder häufiger auch als Konvektions-Diffusionsgleichung bezeichnet wird. Konkret bedeutet dies zum einen, dass der Diffusions-Term Δu explizit mit einem reellen Faktor $\epsilon \in \mathbb{R}_{>0}$ ausgestattet wird und zum anderen, dass der Gleichung ein Konvektionsterm ∇u hinzugefügt wird, welcher einen vektoriellen Faktor $\beta \in \mathbb{R}^m$ erhält. Die Konvektions-Diffusionsgleichung besitzt dann die Form (1.8)

$$-\epsilon \cdot \Delta u + \beta \cdot \nabla u = f \text{ in } \Omega \tag{1.7}$$

$$u = 0 \text{ auf } \partial\Omega. \tag{1.8}$$

Aufgrund der eingeführten Vereinfachung, dass die Gleichungen auf dem $\mathbb{R}^2$ betrachtet werden, resultiert die Konvektions-Diffusionsgleichung mit gleichbleibenden Randbedingungen in Formel (1.9).

$$\beta_1 \cdot \frac{\partial u}{\partial x} + \beta_2 \cdot \frac{\partial u}{\partial y} - \epsilon \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = f \text{ in } \Omega$$
$$u = 0 \text{ auf } \partial\Omega. \tag{1.9}$$

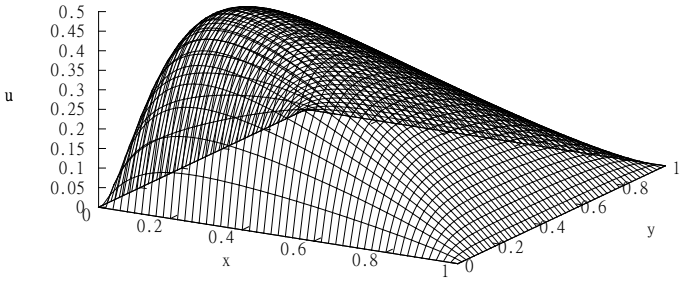


Abbildung 1.3.: Dreidimensionale Visualisierung der diskretisierten Konvektions-Diffusionsgleichung ($n = 50$, $\beta = (-1.0, -1.0)^T$, $\epsilon = 0.1$, $e = 10^{-4}$, $f \equiv 1$)

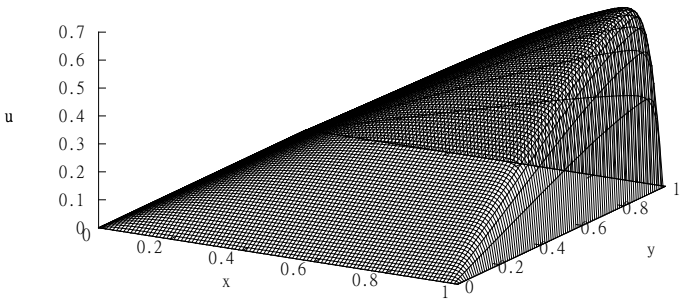


Abbildung 1.4.: Dreidimensionale Visualisierung der diskretisierten Konvektions-Diffusionsgleichung ($n = 100$, $\beta = (1.3, 1.2)^T$, $\epsilon = 0.02$, $e = 10^{-4}$, $f \equiv 1$)

In der Praxis kommen häufig Gleichungen vor, bei denen ϵ sehr klein und die Konvektion hingegen vergleichsweise stark ist (Konvektionsdominanz). Derartig kleine Diffusionsparameter bringen große numerische Probleme mit sich (Vgl. [Bra09]), wie auch später im Verlauf der Arbeit noch erläutert wird.

2. Numerisches Lösen der Konvektions-Diffusionsgleichungen

2.1. Diskretisierung mithilfe der Finite-Differenzen-Methode

2.1.1. Poisson-Gleichung

Da über die gesuchte Funktion u der Konvektions-Diffusionsgleichung im Allgemeinen nicht genügend Informationen vorhanden sind, um diese kontinuierlich bestimmen zu können, muss das Problem zunächst diskretisiert [Hac93, DR06] werden. An dieser Stelle wird dazu die sogenannte Finite-Differenzen-Methode angewendet, d.h. über die Umgebung (welche aufgrund der vereinfachen Annahmen dem Einheitsquadrat im Raum $\mathbb{R}^2$ entspricht) wird zunächst ein geeignetes äquidistantes isotropes Gitter mit der Gitterhöhe h gelegt. Die partiellen Ableitungen werden dann mit Differenzen approximiert, sodass die Funktion u punktweise bestimmt werden kann. Abbildung 2.1 stellt dies dar.

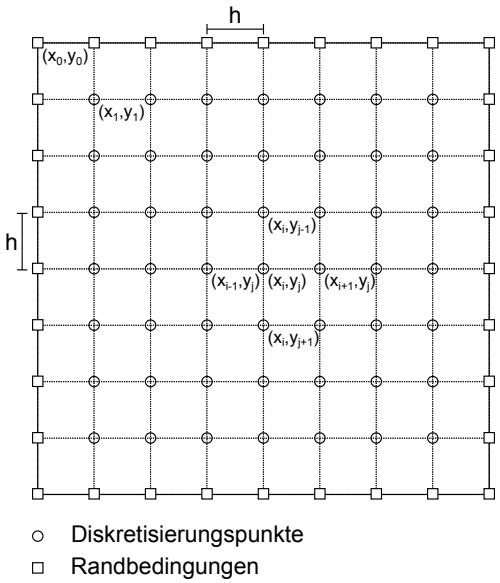


Abbildung 2.1.: Diskretisierung der unbekannten Funktion u mithilfe eines äquidistanten isotropen Gitters

Aufgrund der Einteilung der Umgebung in n Quadrate in jede Richtung ist die Gitterhöhe durch

$$h = n^{-1} \tag{2.1}$$

gegeben. Die Raumkoordinaten der Diskretisierungspunkte lassen sich durch

$$x_i = \frac{i}{n} = i \cdot h \tag{2.2}$$

$$y_j = \frac{j}{n} = j \cdot h \tag{2.3}$$

beschreiben. Es werden also die Werte bestimmt, welche u an den Gitterpunkten (x_i, y_j) für alle $0 \leq i, j \leq n$ einnimmt. Die Werte an diesen Punkten werden mit $u_{i,j}$ bezeichnet, also

$$u_{i,j} = u\left(x_i, y_j\right). \tag{2.4}$$

Um die Poisson-Gleichung bzw. die Konvektions-Diffusionsgleichungen darstellen zu können, muss der Ausdruck Δu in irgendeiner Form approximiert werden. Je nachdem welche Approximation verwendet wird, erhält man unterschiedliche Koeffizientenmatrizen [RXY+00] mit unterschiedlichen numerischen Eigenschaften. Im Folgenden kommt der zentrale Differenzenquotient

$$g'\left(x\right) \approx \frac{g\left(x + \frac{1}{2} \cdot h\right) - g\left(x - \frac{1}{2} \cdot h\right)}{h} \tag{2.5}$$

mit genügend kleinem h zur Anwendung. Wird dieser ein zweites Mal angewendet, so lässt sich damit die zweite Ableitung einer Funktion approximieren:

$$g''\left(x\right) \approx \frac{\frac{g\left(x+\frac{1}{2}\cdot h+\frac{1}{2}\cdot h\right)-g\left(x+\frac{1}{2}\cdot h-\frac{1}{2}\cdot h\right)}{h} - \frac{g(x-\frac{1}{2}\cdot h+\frac{1}{2}\cdot h)-g(x-\frac{1}{2}\cdot h-\frac{1}{2}\cdot h)}{h}}{h} \tag{2.6}$$

$$\Rightarrow g''\left(x\right) \approx \frac{\frac{g(x+h)-g(x)-g(x)+g(x-h)}{h}}{h} \tag{2.7}$$

$$\Rightarrow g''\left(x\right) \approx \frac{g(x+h) + g(x-h) - 2 \cdot g(x)}{h^2} \tag{2.8}$$

Angewendet auf die Funktion u ergibt sich so für die Ableitungen in beide Raumrichtungen folgende Approximationen:

$$\frac{\partial^2 u}{\partial x^2}\left(x_i, y_j\right) \approx \frac{u\left(x_i + h, y_j\right) + u\left(x_i - h, y_j\right) - 2 \cdot u\left(x_i, y_i\right)}{h^2} \tag{2.9}$$

$$\Rightarrow \frac{\partial^2 u}{\partial x^2}\left(x_i, y_j\right) \approx \frac{u\left(x_{i+1}, y_j\right) + u\left(x_{i-1}, y_j\right) - 2 \cdot u\left(x_i, y_i\right)}{h^2} \tag{2.10}$$

$$\Rightarrow \frac{\partial^2 u}{\partial x^2}\left(x_i, y_j\right) \approx \frac{u_{i+1,j} + u_{i-1,j} - 2 \cdot u_{i,j}}{h^2} \tag{2.11}$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_j + h) + u(x_i, y_j - h) - 2 \cdot u(x_i, y_i)}{h^2} \tag{2.12}$$

$$\Rightarrow \frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u(x_i, y_{j+1}) + u(x_i, y_{j-1}) - 2 \cdot u(x_i, y_i)}{h^2} \tag{2.13}$$

$$\Rightarrow \frac{\partial^2 u}{\partial y^2}(x_i, y_j) \approx \frac{u_{i,j+1} + u_{i,j-1} - 2 \cdot u_{i,j}}{h^2} \tag{2.14}$$

Mithilfe dieser beiden Approximationen lässt sich nun auch Δu entsprechend approximieren:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2.15}$$

$$\Rightarrow \Delta u \approx \frac{u_{i+1,j} + u_{i-1,j} - 2 \cdot u_{i,j}}{h^2} + \frac{u_{i,j+1} + u_{i,j-1} - 2 \cdot u_{i,j}}{h^2} \tag{2.16}$$

$$\Rightarrow \Delta u \approx \frac{1}{h^2} \cdot (u_{i+1,j} + u_{i-1,j} - 2 \cdot u_{i,j} + u_{i,j+1} + u_{i,j-1} - 2 \cdot u_{i,j}) \tag{2.17}$$

$$\Rightarrow \Delta u \approx \frac{1}{h^2} \cdot (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4 \cdot u_{i,j}) \tag{2.18}$$

Mithilfe dieser Annäherung verbleibt bei genügend kleinem h nur noch das Gleichungssystem mit den linearen Gleichungen

$$-\frac{1}{h^2} \cdot (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4 \cdot u_{i,j}) = f_{ij} \tag{2.19}$$

$$\Rightarrow \frac{1}{h^2} \cdot (-u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} + 4 \cdot u_{i,j}) = f_{ij} \tag{2.20}$$

$$\Rightarrow \frac{1}{h^2} \cdot (4 \cdot u_{i,j} - u_{i,j+1} - u_{i+1,j} - u_{i-1,j} - u_{i,j-1}) = f_{ij} \tag{2.21}$$

für alle i, j zu lösen, wobei mit f_{ij} analog der Funktionswert von f an der Stelle (x_i, y_j) bezeichnet wird. Aufgrund ihrer Struktur wird Formel (2.21) häufig als Fünfpunktformel bezeichnet.
Obwohl es sich um insgesamt $(n+1)^2$ Gleichungen handelt, reicht es aus $(n-1)^2$ Gleichungen zu betrachten, da aufgrund der Randbedingungen $u_{ij} = 0$ für $i = 0$, $j = 0$, $i = n$ oder $j = n$ gilt und diese Freiheitsgrade so bereits bekannt sind. Obwohl man normalerweise die Randbedingungen in die Koeffizientenmatrix einbeziehen müsste, kann an dieser Stelle aufgrund der homogenen Dirichlet-Randbedingungen darauf verzichtet werden. Die entsprechenden Koeffizienten wären ohnehin Null.

Damit sich das Problem in der herkömmlichen Matrixform

$$A \cdot x = b \tag{2.22}$$

für lineare Gleichungssysteme schreiben lässt, ist es notwendig die Gleichungen für die Unbekannten u_{ij} des zweidimensionalen Problems auf einen einfach indizierten Vektor darzustellen (vgl. [Hac93]). Dabei lassen sich verschiedene Varianten benutzen [Hac93, U.T]:

- Lexikographische Nummerierung
  Hierbei wird das Gitter entweder zeilen- oder spaltenweise beginnend von oben oder unten links durchnummeriert. Für $n = 5$ sieht das etwa folgendermaßen aus:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}, \quad \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} \tag{2.23}$$

- Diagonale Nummerierung:
  "Beginnend vom linken unteren Randpunkt, werden die Punkte so durchnummeriert, daß [sic] sie auf einem nach rechts oben wachsenden, gleichschenkligen Dreieck liegen" [U.T]. Auch dies kann vom linken oberen Randpunkt aus geschehen:

$$\begin{pmatrix} 10 & 13 & 15 & 16 \\ 6 & 9 & 12 & 14 \\ 3 & 5 & 8 & 11 \\ 1 & 2 & 4 & 7 \end{pmatrix} \tag{2.24}$$

- Schachbrettartig:
  Hierbei wird das Gitter in "weiße" und in "schwarze" Felder eingeteilt, die dann nummeriert werden:

$$\begin{pmatrix} 15 & 7 & 16 & 8 \\ 5 & 13 & 6 & 14 \\ 11 & 3 & 12 & 4 \\ 1 & 9 & 2 & 10 \end{pmatrix} \tag{2.25}$$

Während die zeilen- und spaltenweise Nummerierung auf gleiche Matrizenstrukturen führen, ergeben die beiden anderen Nummerierungsarten gänzlich andere Matrizenarten. Abhängig von der Wahl der Nummerierung besitzen die Koeffizientenmatrizen unterschiedliche numerische Eigenschaften.

In dieser Arbeit wird allgemein von einer Nummerierung ausgegangen, die beginnend bei dem oberen linken Element zeilenweise vorgenommen wird. Das bedeutet, es ergibt sich ein lineares Gleichungssystem folgender Form:

$$\frac{1}{h^2} \cdot \begin{pmatrix} \mathbf{T} & -\mathbf{I} & & & & & \\ -\mathbf{I} & \ddots & \ddots & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & \ddots & -\mathbf{I} \\ & & & & & -\mathbf{I} & \mathbf{T} \end{pmatrix} \cdot \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ \vdots \\ u_{1,n-1} \\ u_{2,1} \\ \vdots \\ u_{2,n-1} \\ \vdots \\ u_{n-1,n-1} \end{pmatrix} = \begin{pmatrix} f_{1,1} \\ f_{1,2} \\ \vdots \\ f_{1,n-1} \\ f_{2,1} \\ \vdots \\ f_{2,n-1} \\ \vdots \\ f_{n-1,n-1} \end{pmatrix} \tag{2.26}$$

mit

$$T = \begin{pmatrix} \mathbf{4} & \mathbf{-1} & & & & & \\ \mathbf{-1} & \ddots & \ddots & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & \ddots & \mathbf{-1} \\ & & & & & \mathbf{-1} & \mathbf{4} \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)} \tag{2.27}$$

und I als Einheitsmatrix des Raums $\mathbb{R}^{(n-1)\times(n-1)}$. Für $n = 4$ erhält man beispielsweise mit $f \equiv 1$ folgendes LGS:

$$16 \cdot \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix} \cdot \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{2.28}$$

Zumindest für die diskretisierte Poisson-Gleichung lässt sich der Fehler sehr genau angeben, der bei der hier benutzten Diskretisierungsmethode entsteht. Das Theorem sowie der Beweis dazu lässt sich dem Buch von Dahmen und Reuske ([DR06]) entnehmen.

Theorem 2.1.1 *Es gelte $u \in C^4\left(\overline{\Omega}\right)$. Setze weiterhin*

$$C := \max\left\{ \left\| \frac{\partial^4 u}{\partial x^4} \right\|_{\infty,\overline{\Omega}}, \left\| \frac{\partial^4 u}{\partial y^4} \right\|_{\infty,\overline{\Omega}} \right\}$$

Dann gilt für den bei der Diskretisierung entstehende Fehler $e_h = u_{|\Omega_h} - u_{h|\Omega_h}$ der Poisson-Gleichung folgende Ungleichung:

$$\|e_h\|_\infty \leq \frac{1}{48} \cdot C \cdot h^2$$

Das Problem dieses Theorems ist jedoch die relativ starke Forderung nach der vierten Ableitung, die durchaus für praktische Probleme nicht immer gegeben ist. Ist die Funktion u jedoch viermal stetig differenzierbar, so liegt der Fehler in $\mathcal{O}\left(h^2\right)$.

2.1.2. Konvektions-Diffusionsgleichung

Für den Schritt von der spezielleren Poisson-Gleichung auf die allgemeinere Konvektions-Diffusionsgleichung verändert sich auch die Koeffizientenmatrix. Zum einen muss der reelle Faktor ϵ einbezogen werden. Dazu wird der Faktor $\frac{1}{h^2}$ vor der Matrix durch den Faktor $\frac{\epsilon}{h^2}$ ersetzt. Um den Konvektionsterm ∇u einbeziehen zu können, muss dieser ebenfalls mithilfe eines Differenzenquotienten approximiert werden. Es wird an dieser Stelle zunächst der zentrale Differenzenquotient angewendet. Die Anwendung der einseitigen Quotienten zur Stabilisierung des Verfahrens wird in Abschnitt 5.3 ausgeführt.

$$\frac{\partial u}{\partial x}(x_i, y_j) \approx \frac{u(x_i + h, y_j) - u(x_i - h, y_j)}{2 \cdot h} \tag{2.29}$$

$$\Rightarrow \frac{\partial u}{\partial x}(x_i, y_j) \approx \frac{u_{i+1,j} - u_{i-1,j}}{2 \cdot h} \tag{2.30}$$

$$\frac{\partial u}{\partial y}(x_i, y_j) \approx \frac{u(x_i, y_j + h) - u(x_i, y_j - h)}{2 \cdot h} \tag{2.31}$$

$$\Rightarrow \frac{\partial u}{\partial y}(x_i, y_j) \approx \frac{u_{i,j+1} - u_{i,j-1}}{2 \cdot h} \tag{2.32}$$

$$\beta \cdot \nabla u = \beta_1 \cdot \frac{\partial u}{\partial x} + \beta_2 \cdot \frac{\partial u}{\partial y} \tag{2.33}$$

$$\Rightarrow \beta \cdot \nabla u \approx \beta_1 \cdot \frac{u_{i+1,j} - u_{i-1,j}}{2 \cdot h} + \beta_2 \cdot \frac{u_{i,j+1} - u_{i,j-1}}{2 \cdot h} \tag{2.34}$$

An dieser Stelle wird auch bereits der Nachteil des symmetrischen Differenzenquotienten zum Approximieren des Nabla-Operators sichtbar. Da hier lediglich die erste Ableitung angenähert wird, besitzt diese Diskretisierung keine Stabilität auf den Hauptdiagonaleinträgen.

Das Einbeziehen des Diffusionskoeffizienten und des Konvektionsterms führt zur Systemmatrix (2.35).

$$L_h = \begin{pmatrix} \mathbf{T} & \mathbf{W} & & & & & \\ \mathbf{V} & \ddots & \ddots & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & \ddots & \mathbf{W} \\ & & & & & \mathbf{V} & \mathbf{T} \end{pmatrix} \in \mathbb{R}^{(n-1)^2 \times (n-1)^2} \tag{2.35}$$

mit

$$T = \begin{pmatrix} \frac{4\epsilon}{h^2} & -\frac{\epsilon}{h^2} + \frac{\beta_2}{2h} & & & & & & \\ -\frac{\epsilon}{h^2} - \frac{\beta_2}{2h} & \ddots & \ddots & & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & -\frac{\epsilon}{h^2} + \frac{\beta_2}{2h} & \\ & & & & & -\frac{\epsilon}{h^2} - \frac{\beta_2}{2h} & \frac{4\epsilon}{h^2} & \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)} \tag{2.36}$$

$$V = \begin{pmatrix} -\frac{\epsilon}{h^2} - \frac{\beta_1}{2h} & & & & & & \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & & & \ddots & & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & & -\frac{\epsilon}{h^2} - \frac{\beta_1}{2h} \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)} \tag{2.37}$$

$$W = \begin{pmatrix} -\frac{\epsilon}{h^2} + \frac{\beta_1}{2h} & & & & & & \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & & & \ddots & & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & & -\frac{\epsilon}{h^2} + \frac{\beta_1}{2h} \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)} \tag{2.38}$$

Die Verwendung der symmetrischen Differenzenquotienten führt im Falle der zweidimensionalen Konvektions-Diffusionsgleichung ebenfalls zu einem Diskretisierungsfehler, der in $\mathcal{O}(h^2)$ liegt [Bad01].

2.2. Eigenschaften der Koeffizientenmatrix

Im Folgenden werden einige Eigenschaften der bei der Diskretisierung der Poisson- bzw. Konvektions-Diffusionsgleichung entstehenden Koeffizientenmatrix aufgeführt, die insbesondere in Hinsicht auf numerische Stabilität und Konvergenz der Verfahren wichtig sind, die im Verlauf dieser Arbeit vorgestellt werden.

Schwache Besetztheit Neben den Einträgen auf der Hauptdiagonalen sind in jeder Zeile der Matrix höchstens noch vier weitere von Null verschiedene Einträge zu finden. Im

Falle der vollständigen Koeffizientenmatrix bedeutet das, dass die Matrix zwar insgesamt $(n-1)^4$ Elemente besitzt, von denen jedoch maximal $5 \cdot (n-1)^2$ Nicht-Nulleinträge sind. Die Matrix ist also dünnbesetzt im engeren Sinne, was bedeutet, dass genügend Nicht-Nulleinträge existieren, dass es sich bereits lohnt, diese Eigenschaft von Algorithmen und Datenstrukturen ausnutzen zu lassen (vgl. [DM11]). In Kapitel 3 wird genauer erläutert, wie sich diese Eigenschaft verwenden lässt, um sowohl einen geringen Speicherbedarf als auch eine erhöhte Ausführungsgeschwindigkeit zu erzielen.

Irreduzibilität Die Koeffizientenmatrix besitzt die Eigenschaft der Irreduzibilität. Dies kann man sich leicht anhand des Belegungsgraphen der Matrix überlegen, da die Hauptdiagonale vollständig belegt ist. Alle weiteren Einträge innerhalb der Matrix sind paarweise über die Fünfpunktformel miteinander verbunden.

Nicht-Negativer Typ Folgende Definition zum nicht-negativen Typ einer Matrix entstammt dem Finite-Elemente-Vorlesungsskriptum von Professor Braack ([Bra09]).

Definition 2.2.1 (Nicht-Negativer Typ) *Eine quadratische Matrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ heißt: [...] vom nicht-negativen Typ, wenn alle Nebendiagonalelemente nicht-positiv und die Diagonalelemente positiv sind.*

Theorem 2.2.2 *Die Systemmatrix der diskretisierten Konvektions-Diffusionsgleichung wie in 2.35 aufgeführt ist im Falle*

$$h < \frac{2 \cdot \epsilon}{\|\beta\|_1}$$

vom nicht-negativen Typ.

Beweis Theorem 2.2.2 Die Hauptdiagonaleinträge sind wegen $\epsilon, h > 0$ stets positiv. Für den Nachweis des nicht-negativen Typs genügt es zu zeigen, dass die von Null verschiedenen Nebendiagonaleinträge a_{ij} negativ sind. Diese Koeffizienten sind von der Form

$$a_{ij} = -\frac{\epsilon}{h^2} - \frac{b}{2 \cdot h} \tag{2.39}$$

mit einem $b \in \mathbb{R}$, $|b| < \|\beta\|_1$. Für diese gilt

$$a_{ij} = -\frac{\epsilon}{h^2} - \frac{b}{2 \cdot h} \tag{2.40}$$

$$\Rightarrow a_{ij} < -\frac{\epsilon}{h^2} + \frac{\|\beta\|_1}{2 \cdot h} \tag{2.41}$$

$$\Rightarrow a_{ij} < -\frac{\epsilon}{h^2} + \frac{\frac{2 \cdot \epsilon}{h}}{2 \cdot h} \tag{2.42}$$

$$\Rightarrow a_{ij} < 0 \tag{2.43}$$

Die Matrix ist somit vom nicht-negativen Typ.

□

Diagonaldominanz Die Koeffizientenmatrix der Konvektions-Diffusionsgleichung ist unter bestimmten Voraussetzungen an die Koeffizienten erweitert diagonaldominant. Eine genaue Definition zur erweiterten Diagonaldominanz lässt sich ebenfalls [Bra09] entnehmen:

Definition 2.2.3 (Erweiterte Diagonaldominanz) *Eine quadratische Matrix* $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ *heißt: [...] erweitert diagonaldominant, wenn*

$$|a_{ii}| \geq \sum_{i\neq j} |a_{ij}| \;\; \forall i \in \{1,\ldots,n\}$$
$$|a_{ii}| > \sum_{i\neq j} |a_{ij}| \;\; \textit{für mindestens ein } i \in \{1,\ldots,n\}$$

Theorem 2.2.4 *Die Systemmatrix der diskretisierten Konvektions-Diffusionsgleichung wie in 2.35 aufgeführt ist im Falle*

$$h < \frac{2\cdot\epsilon}{\|\beta\|_1}$$

erweitert diagonaldominant.

Beweis Theorem 2.2.4 Ein Punkt innerhalb des Diskretisierungsgitters kann entweder in einer der vier Ecken, an einem der vier Ränder oder randfern liegen. Aus diesem Grund müssten insgesamt neun verschiedene Fälle unterschieden werden. Tatsächlich ist es jedoch ausreichend den pessimalen Fall zu betrachten (randferner Punkt), da damit auch für alle anderen Fälle die schwache Diagonaldominanz gezeigt ist. Es werden hier lediglich die Zeilensummen betrachtet, aus denen dann aufgrund des nicht-negativen Typs der Matrix die schwache bzw. starke Diagonaldominanz direkt gefolgert werden kann.

$$\sum_{j=1}^{n} a_{ij} = \left(-\frac{\epsilon}{h^2} - \frac{\beta_1}{2h}\right) + \left(-\frac{\epsilon}{h^2} - \frac{\beta_2}{2h}\right) + \left(-\frac{\epsilon}{h^2} + \frac{\beta_1}{h}\right) + \left(-\frac{\epsilon}{h^2} + \frac{\beta_2}{h}\right) + \left(\frac{4\cdot\epsilon}{h^2}\right) \tag{2.44}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} = -\frac{4\cdot\epsilon}{h^2} + \frac{4\cdot\epsilon}{h^2} \tag{2.45}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} = 0 \tag{2.46}$$

Für die erweiterte Diagonaldominanz ist es notwendig für zumindest eine Zeile die strikte Diagonaldominanz nachzuweisen. Hierzu wird die erste Zeile der Matrix betrachtet.

$$\sum_{j=1}^{n} a_{ij} = \left(-\frac{\epsilon}{h^2} + \frac{\beta_2}{2h}\right) + \left(-\frac{\epsilon}{h^2} + \frac{\beta_1}{2h}\right) + \left(\frac{4 \cdot \epsilon}{h^2}\right) \tag{2.47}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} = -\frac{2\epsilon}{h^2} + \frac{\beta_1 + \beta_2}{2h} + \frac{4 \cdot \epsilon}{h^2} \tag{2.48}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} = \frac{2\epsilon}{h^2} + \frac{\beta_1 + \beta_2}{2h} \tag{2.49}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} > \frac{2 \cdot \frac{1}{2} \cdot h \cdot \|\beta\|_1}{h^2} + \frac{\beta_1 + \beta_2}{2h} \tag{2.50}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} > \frac{\|\beta\|_1}{h} + \frac{\beta_1 + \beta_2}{2h} \tag{2.51}$$

$$\Rightarrow \sum_{j=1}^{n} a_{ij} > 0 \tag{2.52}$$

□

M-Matrix Die Eigenschaft einer M-Matrix ist für die numerische Betrachtung sehr wichtig, da diese später die Existenz der ILU-Zerlegung garantieren wird. Folgende Definition einer M-Matrix entstammt der Dissertation von Bader ([Bad01]).

Definition 2.2.5 (M-Matrix) *Eine Matrix $A = (a_{ij})$ heißt M-Matrix, falls folgende Eigenschaften erfüllt sind:*

- $a_{ij} \leq 0$ *für* $i \neq j$
- A *ist nicht-singulär*
- $A^{-1} \geq 0$

Folgender Satz aus [Hac93] gibt die Möglichkeit, die Eigenschaft der M-Matrix zu zeigen, ohne die obigen Eigenschaften zuerst explizit nachweisen zu müssen.

Theorem 2.2.6 *Sei $A \in \mathbb{R}^{n \times n}$ eine Matrix, für die folgende Eigenschaften gelten:*

- A *ist stark oder irreduzibel diagonal dominant*
- A *ist vom nicht-negativen Typ.*

Dann ist A eine M-Matrix.

Das bedeutet insbesondere mit den oben aufgeführten Eigenschaften, dass die Koeffizientenmatrix mit der Bedingung

$$h < \frac{2 \cdot \epsilon}{\|\beta\|_1} \tag{2.53}$$

bereits die M-Matrix-Eigenschaft besitzt. Die irreduzible Diagonaldominanz bedeutet lediglich, dass die Matrix irreduzibel ist (dies gilt immer) und zudem erweitert diagonaldominant ist (dies folgt direkt aus den gegebenen Bedingungen). Mit den gleichen Bedingungen an die Koeffizienten ist die Matrix zudem vom nicht-negativen Typ.

Nach Kontraposition folgt an dieser Stelle auch die Regularität der Matrix in diesem Falle, was überhaupt die Lösbarkeit des LGS bedeutet. Abbildung 2.2 zeigt eine fehlerhafte Berechnung aufgrund falscher Koeffizienten. Es lässt sich deutlich erkennen, dass die Berechnung oszilliert, statt gegen die korrekte physikalische Lösung zu konvergieren.

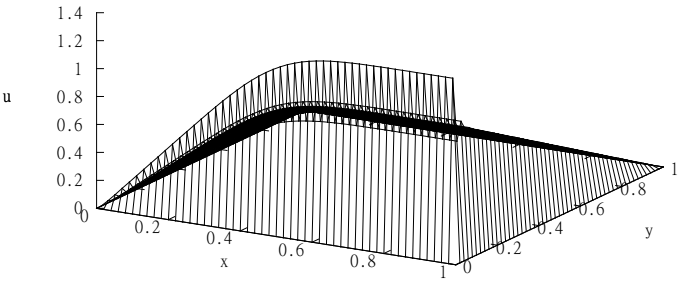


Abbildung 2.2.: Dreidimensionale Visualisierung der Konvektions-Diffusionsgleichung ($n = 50$, $\beta = (0.5, -1.0)^T$, $\epsilon = 0.005$, $e = 10^{-4}$, $f \equiv 1$).

Konditionierung Zumindest für die Koeffizientenmatrix L_h der diskretisierten Poisson-Gleichung lässt sich an dieser Stelle die Konditionszahl sehr genau angeben. Wie man leicht ablesen kann, gilt zum einen $\|L_h\|_\infty \leq 8 \cdot h^{-2}$, zum anderen ist nachweisbar, dass die Ungleichung $\left\|L_h^{-1}\right\|_\infty \leq \frac{1}{8}$ gilt [Hac86]. Das führt insgesamt zu einer Konditionszahl von

$$\operatorname{cond}_\infty(L_h) = \|L_h\|_\infty \cdot \left\|L_h^{-1}\right\|_\infty \tag{2.54}$$

$$\Rightarrow \operatorname{cond}_\infty(L_h) \leq 8 \cdot h^{-2} \cdot \frac{1}{8} \tag{2.55}$$

$$\Rightarrow \operatorname{cond}_\infty(L_h) \leq h^{-2} \tag{2.56}$$

Die Konditionszahl wächst also quadratisch mit der Auflösung des Diskretisierungsgitters.

Symmetrie Im Falle der speziellen Poisson-Gleichung ist die Systemmatrix L_h symmetrisch, im allgemeinen Fall der Konvektions-Diffusionsgleichung nur dann, wenn der Konvektions-Term wegfällt.

Konvergenzbetrachtung Obwohl in dieser Arbeit lediglich die Richardson-Iteration als Lösungsverfahren betrachtet wird, wird an dieser Stelle kurz darauf eingegangen, wie sich andere (einfache) iterative Verfahren bezüglich der Systemmatrix der Diskretisierung verhalten. Die Theoreme 2.2.7 (aus [Stü05]) und 2.2.8 (aus [Bra05]) geben Bedingungen für die Konvergenz zweier Verfahren an.

Theorem 2.2.7 *Ist eine Matrix $A \in C^{n\times n}$ unzerlegbar (gemeint ist die Irreduzibilität, N.C.E.) und erfüllt sie das schwache Zeilensummenkriterium (gemeint ist die erweiterte Diagonaldominanz, N.C.E.), so konvergiert das Jacobi-Verfahren bei beliebigem Startvektor $x^0 \in C^n$ für beliebige rechte Seite $b \in C^n$ gegen $A^{-1}b$.*

Theorem 2.2.8 *Wenn A positive Elemente in der Hauptdiagonale hat und alle andern [sic] Elemente ≤ 0 sind, dann konvergiert das Gauß-Seidel-Verfahren genau dann, wenn das Jacobi-Verfahren konvergiert. Wenn beide Verfahren konvergieren, dann ist das Gauß-Seidel-Verfahren asymptotisch schneller.*

Wie bereits weiter oben festgestellt wurde, ist die Irreduziblität immer gegeben, die erweiterte Diagonaldominanz und der nicht-negative Typ der Systemmatrix hingegen im Falle von

$$h < \frac{2 \cdot \epsilon}{\|\beta\|_1}. \tag{2.57}$$

Das bedeutet insbesondere , dass sowohl das Jacobi- als auch das Gauß-Seidel-Verfahren in diesem Fall bereits konvergieren. Die Konvergenzgeschwindigkeit strebt jedoch im Falle der diskretisierten Konvektions-Diffusionsgleichung gegen $1 - \mathcal{O}(h^2)$. Bei hinreichend hoher Auflösung des Diskretisierungsgitters liegt lediglich eine lineare Konvergenzrate vor [Sue08].

2.3. Die Richardson-Iteration

Lineare Iterationsverfahren zur Lösung von linearen Gleichungssystemen $Ax = b$ haben allgemein die Form [PKA03]

$$\Phi(x) := Mx + Nb \tag{2.58}$$

mit Matrizen M und N, die der Koeffizientenmatrix A entsprechend gewählt werden. Für eine einfache Form des Iterationsverfahren wird $N = I$ gewählt, welches dann als stationäres Richardson-Verfahren [Hac93] oder einfach nur als Richardson-Iteration bezeichnet wird. Um die Konsistenz des Verfahrens mit dem linearen Gleichungssystem $Ax = b$ zu gewährleisten, müssen die Matrizen die Gleichung $I = M + NA$ erfüllen. Die

feste Wahl von N führt demnach zur folgenden Wahl von M:

$$I = M + NA \tag{2.59}$$
$$\Leftrightarrow I = M + A \tag{2.60}$$
$$\Leftrightarrow M = I - A \tag{2.61}$$

Ein Iterationsschritt mit dem Richardson-Verfahren lässt sich so folgendermaßen beschreiben:

$$x_{n+1} = (I - A)\, x_n + b \tag{2.62}$$
$$\Leftrightarrow x_{n+1} = x_n - Ax_n + b \tag{2.63}$$
$$\Leftrightarrow x_{n+1} = x_n - (Ax_n - b) \tag{2.64}$$

Theorem 2.3.1 *Sei A eine Matrix die nur positive Eigenwerte besitzt und $\lambda_{min}(A)$, $\lambda_{max}(A)$ der minimale bzw. maximale Eigenwert der Matrix. Dann konvergiert das obige Verfahren genau dann, wenn die Ungleichung*

$$\lambda_{max}(A) < 2 \tag{2.65}$$

erfüllt ist.

Weiterhin ist in diesem Fall die Konvergenzrate des Iterationsverfahrens gegeben durch

$$\rho_{Rich} = \max\left\{\left|1 - \lambda_{min}(A)\right|, \left|1 - \lambda_{max}(A)\right|\right\} \tag{2.66}$$

Obwohl dieses Verfahren in der Berechnung keinen großen Aufwand benötigt, so ist die Konvergenzeigenschaft, die Theorem 2.3.1 fordert (der Beweis lässt sich [Hac93] entnehmen), im Allgemeinen nicht für die in den vorherigen Abschnitten eingeführten Koeffizientenmatrizen gegeben. Aus diesem Grund ist an dieser Stelle ein Vorkonditionierer notwendig, welcher durch die inkomplette LR-Zerlegung realisiert ist. Diese Zerlegung und die Anwendung als Vorkonditionierer wird im folgenden Abschnitt beschrieben.

Wie sich das Verfahren noch zusätzlich modifizieren lässt, um beispielsweise mit einem Dämpfungsparameter bessere Konvergenz zu erhalten, wird in Abschnitt 5.1 erläutert.

2.4. Die ILU-Zerlegung

Bei der gewöhnlichen LR- oder auch LU-Zerlegung wird eine gegebene Koeffizientenmatrix A beispielsweise per gaußschem Eliminationsverfahren in eine untere Dreiecksmatrix L und in eine obere Dreiecksmatrix U zerlegt, sodass $L \cdot U = A$ gilt, beziehungsweise $L \cdot U = P \cdot A$, sofern für die Zeilenvertauschungen zur Stabilisierung eine Permutationsmatrix P notwendig ist. Eine derartige Zerlegung wird häufig für direkte Löser eingesetzt, bei denen mehrere lineare Gleichungssysteme zu lösen sind, die zwar gleiche Koeffizientenmatrizen, jedoch unterschiedliche rechte Seiten besitzen. Der etwas höhere, kubische Aufwand der LU-Zerlegung amortisiert sich dann schnell, da das Gleichungssystem mit der Zerlegung etwas effizienter in quadratischer Laufzeit per Vorwärts-/Rückwärtseinsetzen gelöst werden kann [HD08].

Die LU-Zerlegungen haben jedoch im Allgemeinen nicht die gleiche Besetzungsstruktur wie die Ursprungsmatrix, was insbesondere ein Nachteil bei dünnbesetzten Matrizen darstellt. Die resultierende LU-Zerlegung hat häufig deutlich mehr von Null verschiedene Einträge als die zu zerlegende Matrix, sodass das Speichern der Zerlegung deutlich aufwändiger sein kann [Hac93]. Liegt also beispielsweise bereits ein Datentyp zur effizienten Speicherung und Bearbeitung einer Matrix vor, so kann der gleiche Datentyp nicht für die Zerlegung weiterverwendet werden. Ein Beispiel der LU-Zerlegung lässt sich Abbildung 2.3 entnehmen (Das Beispiel ist [Sch11] entnommen).

$$A = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix}, \; LU = \begin{pmatrix} 2 & 1 & 0 & 1 \\ \frac{1}{2} & \frac{3}{2} & 1 & -\frac{1}{2} \\ 0 & \frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \\ \frac{1}{2} & -\frac{1}{3} & \frac{1}{2} & \frac{3}{2} \end{pmatrix}$$

Abbildung 2.3.: Eine Beispielmatrix und deren LU-Zerlegung. Die Einsen auf der Hauptdiagonalen der L-Matrix werden nicht gespeichert.

Aus diesem Grund bietet sich die LR-Zerlegung für die Konvektions-Diffusionsgleichungen nicht an (Hierbei sei insbesondere auf die in Abschnitt 2.2 erwähnte schwache Besetztheit verwiesen). Es wird stattdessen auf eine sogenannte unvollständige LU-Zerlegung [MW06, MV] (engl. incomplete lower-upper decomposition) - im Folgenden als ILU-Zerlegung bezeichnet - zurückgegriffen, die hier als Vorkonditionierer eingesetzt wird. Dazu wird die Koeffizientenmatrix A so in zwei Dreiecksmatrizen L und U zerlegt, sodass lediglich noch $LU \approx A$ bzw. $A = L \cdot U - R$ mit einer gewissen Restmatrix R gilt [Hac93]. Die Dreiecksmatrizen besitzen dort Nulleinträge, wo auch die Ursprungsmatrix A Nulleinträge besitzt. Es gilt also etwas genauer

$$L_{ij} = U_{ji} = 0 \text{ für } 1 \leq i < j \leq n \tag{2.67}$$

und

$$L_{ij} = U_{ij} = 0 \text{ für } (i,j) \notin E(A). \tag{2.68}$$

Am Beispiel der Matrix aus Abbildung 2.3 ergibt sich so eine ILU-Zerlegung wie in Abbildung 2.4 aufgeführt. Diese inkomplette LR-Zerlegung besitzt insgesamt 7 Nulleinträge, während die gewöhnliche LR-Zerlegung lediglich 2 Nulleinträge besitzt.

$$ILU = \begin{pmatrix} 2 & 1 & 0 & 1 \\ \frac{1}{2} & \frac{3}{2} & 1 & 0 \\ 0 & \frac{3}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{3}{2} \end{pmatrix}$$

Abbildung 2.4.: ILU-Zerlegung der Beispielmatrix. Die Einsen auf der Hauptdiagonalen der L-Matrix werden nicht gespeichert.

Die ILU-Zerlegung für eine gegebene Matrix lässt sich beispielsweise durch Algorithmus (2.1) bestimmen, wie er [Beb11] entnommen wurde. Alternativ findet sich in dem Buch von Hackbusch [Hac93] ein anderer Algorithmus, mit dem die Matrix zerlegt werden kann.

Algorithmus 2.1 ILU-Zerlegung

Vorbedingung:

- A ist eine Matrix, zu der eine ILU-Zerlegung existiert.

Nachbedingung:

- A beinhaltet die Matrizen L und U der ILU-Zerlegung von $\overline{A}$, wobei die Einsen von L auf der Hauptdiagonalen nicht gespeichert werden.

---

for $k = 1 \to n - 1$ **do**
  for $i = k + 1 \to n$ **do**
    if $A_{i,k} \neq 0$ **then**
      $A_{i,k} \leftarrow \frac{A_{i,k}}{A_{k,k}}$
      for $j = k + 1 \to n$ **do**
        if $A_{i,j} \neq 0$ **then**
          $A_{i,j} \leftarrow A_{i,j} - A_{i,k} \cdot A_{k,j}$
        end if
      end for
    end if
  end for
end for

Bei bekannter ILU-Zerlegung wird das lineare Gleichungssystem $A \cdot x = b$ vorkonditioniert, indem formal der Ausdruck $(LU)^{-1}$ auf beiden Seiten von links dazumultipliziert wird.

$$Ax = b \tag{2.69}$$

$$\Leftrightarrow \underbrace{(LU)^{-1} A}_{=:\tilde{A}} x = \underbrace{(LU)^{-1} b}_{=:\tilde{b}} \tag{2.70}$$

Die Vorkonditionierung sorgt dafür, dass sich die neue Koeffizientenmatrix $\tilde{A}$ näher an der Einheitsmatrix befindet als A und somit unter Umständen das Konvergieren der Richardson-Iteration überhaupt erst ermöglicht.

Der Ausdruck $(LU)^{-1}$ wird tatsächlich während des Verfahrens nicht berechnet, da dies zu viel Aufwand bedeuten würde. Stattdessen lässt sich die Gleichung $(LU)^{-1} y = z$ effizient per Vorwärts-/Rückwärtseinsetzen nach z auflösen, wie sich Algorithmus (2.2) entnehmen lässt.

Algorithmus 2.2 Auflösen von $(LU)^{-1} y = z$ nach z

Vorbedingung:

- L ist eine untere, U eine obere Dreiecksmatrix.
- y ist ein Vektor.

Nachbedingung:

- Der zurückgelieferte Vektor z ist das Ergebnis von $(LU)^{-1} \cdot y$.

- Löse den Ausdruck $L \cdot r = y$ per Vorwärtseinsetzen nach r auf.
- Löse den Ausdruck $U \cdot z = r$ per Rückwärtseinsetzen nach z auf.
- Liefere z zurück.

Mit den Ergebnissen aus dem vorherigen Abschnitt lässt sich das Verfahren zur iterativen Lösung eines LGS durch Algorithmus (2.3) lösen.

Algorithmus 2.3 Richardson-Iteration mit ILU-Zerlegung als Vorkonditionierer

Vorbedingung:

- A ist eine Koeffizientenmatrix, zu der eine ILU-Zerlegung existiert.
- Die Richardson-Iteration konvergiert für die gegebene Matrix und deren ILU-Zerlegung.
- e ist eine geeignete, nicht-negative Schranke.
- x_0 ist ein geeigneter Startwert.

Nachbedingung:

- Für den zurückgelieferten Vektor x_n gilt $\|A \cdot x_n - b\|_\infty < e$.

- Bilde die ILU-Zerlegung und speichere die Matrizen L und U.
- Beginne mit dem Startwert x_0 und iteriere:

$$x_n = x_{n-1} - \left(\tilde{A}x_{n-1} - \tilde{b}\right)$$

- Falls $\|Ax_n - b\|_\infty < e$ so liefere x_n zurück. Ansonsten wiederhole die Iteration mit dem neuen x_n.

Es stellt sich nun allerdings die Frage, ob die ILU-Zerlegung überhaupt für eine gegebene Koeffizientenmatrix existiert. Deren Existenz ist zumindest für M-Matrizen gesichert [Joh98, MV, Hac93], womit an dieser Stelle auch auf Abschnitt 2.2 verwiesen sei, da die Systemmatrix mit bestimmten Voraussetzungen an die Koeffizienten eine solche Matrix ist.

3. Implementierung des Verfahrens

3.1. Allgemeines

Das Programm zur Aufstellung, Lösung und Visualisierung der Konvektions-Diffusionsgleichungen wurde in der Programmiersprache C++ mithilfe der Klassenbibliothek Qt (http://qt.nokia.com/) implementiert. Da der Schwerpunkt auf eine schnelle Ausführung gelegt wurde, wurde zum einen auf die vollständige Behandlung aller möglicher Fehler und fehlerhaften Eingaben, sowieso auf modularen und wiederverwendbaren Code nach Software-Engineering-Maßstäben verzichtet. Stattdessen wurden in dem Programm sehr spezifische Datenstrukturen und sehr spezifischer Code verwendet. Abbildung 3.1 gibt einen groben Überblick über den Aufbau des Programms.

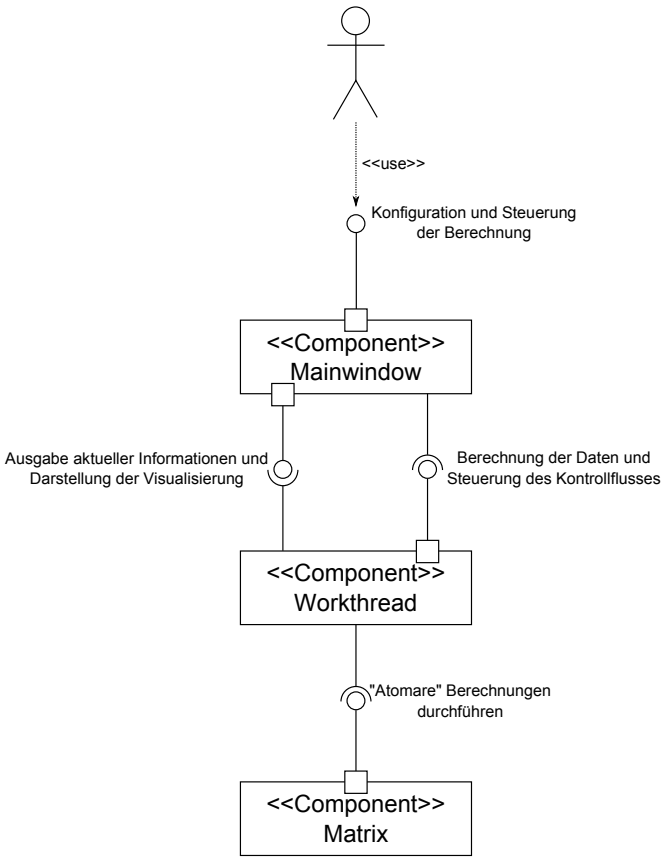


Abbildung 3.1.: Komponentendiagramm des Programms

Die GUI - ebenfalls mithilfe der Qt-Widgets erstellt - ist rudimentär, erlaubt die

Eingabe der Parameter und die Steuerung der Berechnung. Hierbei ist es möglich die folgenden Parameter zu modifizieren:

- Die Auflösung des Diskretisierungsgitters n
- Den Diffusionskoeffizienten ϵ
- Die Richtung und Geschwindigkeit der Konvektion $\beta = (\beta_1, \beta_2)^T$
- Die Genauigkeit e, bis zu der gerechnet werden soll
- Den Dämpfungsparameter ω für die Richardson-Iteration (Abschnitt 5.1)
- Den Parameter γ für das Upwinding, um die Matrix entsprechend über- oder unterrelaxieren zu lassen (Abschnitt 5.1)
- Die Auswahl zwischen der Diskretisierung mit dem symmetrischen Differenzenquotienten und der Upwinding-Diskretisierung (Abschnitt 5.3)

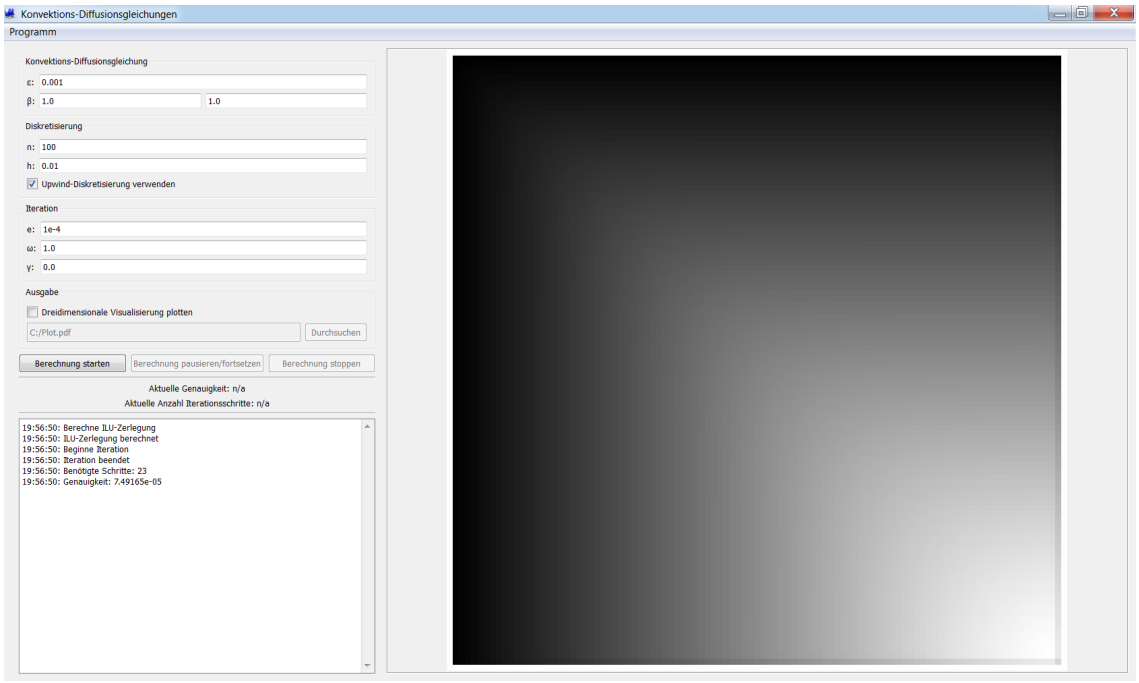


Abbildung 3.2.: Die graphische Benutzeroberfläche des Programms

Weitere Parameter, wie beispielsweise den Startwert x_0 der Iteration oder die Funktion f, lassen sich nicht konfigurieren (insbesondere da die Konfiguration von f einen Funktionsinterpreter benötigen würde, der unter Umständen den Rahmen der Arbeit sprengen würde). Stattdessen wird $f \equiv 1$ und $x_0 \equiv 0$ angenommen. Neben der Möglichkeit die Berechnung vorzunehmen, ermöglicht das Programm, dass das Ergebnis, d.h. die diskret berechnete Funktion u, zweidimensional dargestellt wird. Eine dreidimensionale

Berechnung kann auf Wunsch mithilfe des Programms Gnuplot gezeichnet und in eine beliebige Datei geschrieben werden.

Aufgrund der Funktionsweise der hier verwendeten Algorithmen ist eine Parallelisierung zur Berechnungsbeschleunigung nur schwerlich möglich. So ist etwa die Iteration ebenso wie die ILU-Zerlegung streng sequentiell. Die etwas atomareren Berechnungen wie beispielsweise die Matrix-Vektor-Multiplikation oder das Vorwärts-/Rückwärtseinsetzen ließen sich zwar theoretisch aufteilen, jedoch wäre der hier entstehende Overhead größer als der Geschwindigkeitsgewinn. Um dennoch einen leichten Geschwindigkeitsgewinn zu ermöglichen, wird das iterative Verfahren zeitweise aufgesplittet. Hat das Programm etwa soeben den Ergebnisvektor x_n im Iterationsverfahren berechnet, so wird die Berechnung des Residuums $\|A \cdot x_n - b\|_\infty$ asynchron gestartet, während bereits der nächste Iterationsschritt durchgeführt wird. Bevor jedoch die letzte, finale Berechnung im Iterationsschritt vorgenommen wird, wird dann auf die Berechnung des Residuums gewartet. Ist die gewünschte Genauigkeit bereits erreicht, so bricht das Verfahren ab. Ist die Schranke noch nicht erreicht, so wird der letzte Berechnungsschritt ausgeführt und das Programm erhält den Ergebnisvektor x_{n+1}. Dies ermöglicht zwar bei beispielsweise zwei Rechenkernen keine hundertprozentige Auslastung, jedoch eine von etwa 70 bis 80 Prozent.

3.2. Wahl einer geeigneten Datenstruktur

In diesem Abschnitt wird die Datenstruktur beschrieben, mit der die Koeffizientenmatrix des linearen Gleichungssystems abgespeichert wird. Dies ist theoretisch eigentlich nicht notwendig, da sich die einzelnen Einträge durchaus berechnen lassen und die Matrix strukturiert ist, jedoch sorgt das Zwischenspeichern aufgrund der stark reduzierten Anzahl von Funktionsaufrufen für eine deutlich höhere Ausführungsgeschwindigkeit. Zudem ist es notwendig zumindest die ILU-Zerlegung (welche die gleiche Besetzungsstruktur wie die Koeffizientenmatrix besitzt) abzuspeichern, da diese zu aufwendig zu berechnen ist. An dieser Stelle sei noch bemerkt, dass die Matrix lediglich in einfacher Genauigkeit (32 Bit) abgespeichert werden muss, das Residuum bei der Berechnung hingegen in doppelter Genauigkeit (64 Bit).

Als naiver Ansatz würde es sich anbieten, die Matrix als zweidimensionales Array der Größe $(n-1)^2 \times (n-1)^2$ zu speichern, was einen Speicherbedarf in $\mathcal{O}(n^4)$ bedeuten würde. Bei der Verwendung von einfacher Genauigkeit und unter Vernachlässigung der Pointer wären bei $n = 201$ beispielsweise bereits

$$(201-1)^2 \cdot (201-1)^2 \cdot 32 \ [Bit] \approx 5.96 \ [GiByte] \tag{3.1}$$

notwendig. Für die Praxis ist dieser Ansatz also nicht tauglich.

Das Speichern der Matrix wurde durch einen effizienteren Ansatz vorgenommen. Statt sämtliche Einträge zu speichern, wird die Dünnbesetztheit der Matrix (Abschnitt 2.2) ausgenutzt und lediglich die von Null verschiedenen Einträge gespeichert. Dazu wird

weiterhin ausgenutzt, dass sich sowohl in jeder Zeile, als auch in jeder Spalte der Matrix maximal fünf Einträge befinden. Zum schnellen Zugriff sowohl zeilen- als auch spaltenweise wurden somit zwei Arrays der Größen $(n-1)^2 \times 5$ verwendet, welche jeweils auf die gleichen Einträge verweisen. Die Einträge beinhalten dann sowohl ihre Zeile als auch ihre Spalte (je nachdem von welchem der beiden Zugriffsarrays darauf zugegriffen wird) und den in einfachen Genauigkeit gespeicherten Wert innerhalb der Matrix. Die Erzeugung der Datenstruktur lässt sich dem folgenden Code entnehmen.

```
1  Matrix::Matrix(int n) {
2    /* Speichere die Abmessungen der Matrix */
3    this->n = n;
4    this->dim = (n - 1) * (n - 1);
5
6    /* Initialisiere die Zugriffsfelder */
7    this->rowAcc = new Entry**[dim];
8    this->colAcc = new Entry**[dim];
9
10   for (int i = 0; i < dim; i++) {
11     this->rowAcc[i] = new Entry*[5];
12     this->colAcc[i] = new Entry*[5];
13
14     for (int j = 0; j < 5; j++) {
15       this->rowAcc[i][j] = 0;
16       this->colAcc[i][j] = 0;
17     }
18   }
19 }
```

Listing 3.1: Matrix-Konstruktor

Auch hier wäre es eigentlich nicht notwendig, dass Zeile und Spalte in jedem Eintrag gespeichert wird. Dies wird jedoch ebenfalls aus Gründen der Optimierung vorgenommen. Unter der Voraussetzung, dass 32-Bit-kompatibler Code erzeugt wird, sind die Pointer jeweils 32 Bit groß. Da jeder Eintrag mithilfe zweier Ganzzahlen (32 Bit) die Zeile und die Spalte speichert, sowie mit einfacher Genauigkeit (32 Bit) den Eintrag, sind pro Eintrag 96 Bit zu berechnen.

Für die Arrays in den Zeilen 7 und 8 sind $2 \cdot (n-1)^2 \cdot 32$ Bit notwendig. Die Arrays in den Zeilen 11 und 12 (von denen $(n-1)^2$ erzeugt werden) benötigen zusammen $2 \cdot 5 \cdot 32$ Bit. Da die Zeilen und Spalten in der Matrix jeweils auf identische Einträge verweisen sind zusätzlich $5 \cdot (n-1)^2 \cdot 96$ Bit für eben diese Einträge zu berechnen. Bei $n = 201$ ergibt sich so ein ungefährer Speicherbedarf von

$$2 \cdot (201-1)^2 \cdot 32\ [Bit] + (201-1)^2 \cdot 2 \cdot 5 \cdot 32\ [Bit] + 5 \cdot (201-1)^2 \cdot 96\ [Bit] \approx 4.11\,[MiByte] \tag{3.2}$$

Selbst unter Berücksichtigung der Pointer verbraucht diese Struktur also deutlich weniger Speicherplatz. Wichtiger als der genaue Speicherbedarf an dieser Stelle ist jedoch, dass sich der Speicherbedarf in einer gänzlich anderen Größenordnung befindet als noch zuvor. Der naive Ansatz benötigt Speicherbedarf in der Ordnung von $\mathcal{O}(n^4)$, die modifizierte Datenstruktur hingegen nur noch Speicher in der Ordnung $\mathcal{O}(n^2)$. An dieser Stelle

sei noch einmal daran erinnert, dass die Matrix zwar einen großen Anteil des Speicherbedarfs darstellt, jedoch nicht den alleinigen. Neben der Koeffizientenmatrix wird auch die ILU-Zerlegung sowie diverse Vektoren (deren Speicherbedarf sich ebenfalls in $\mathcal{O}(n^2)$ befinden) abgespeichert. Zusätzlich entstehen während der Berechnung weitere temporäre Datenstrukturen und Overhead, die ebenfalls Speicher benötigen.

Allerdings bietet die neue Datenstruktur nicht nur Vorteile im Bezug auf den Speicherbedarf, sondern auch im Bezug auf die Ausführungsgeschwindigkeit mit sich. So müssen beispielsweise nicht mehr länger sämtliche Nulleinträge in den Algorithmen durchlaufen werden. Stattdessen können effizient sowohl zeilen- als auch spaltenweise lediglich die von Null verschiedenen Einträge durchlaufen werden. Dies senkt an vielen Stellen die Laufzeit ebenfalls von $\mathcal{O}(n^4)$ auf $\mathcal{O}(n^2)$, wie im folgenden Abschnitt genauer erläutert wird.

3.3. Algorithmen

Nachdem im vorherigen Abschnitt der Aufbau der Datenstruktur für die Matrix erläutert wurde, werden an dieser Stelle die Implementierung wichtiger Codebereiche genauer dargestellt.

Matrix-Vektor-Multiplikation Bei der gewöhnlichen Matrix-Vektor-Multiplikation $A \cdot x = c$ wird für gewöhnlich jede Komponente des Ergebnisvektors durch $c_i = \sum_{j=1}^{n} a_{ij} \cdot x_j$ berechnet. Würde man an dieser Stelle auch sämtliche Nulleinträge der Matrix durchlaufen, so ergäbe sich pro Komponente ein quadratischer Aufwand in Abhängigkeit der Auflösung des Diskretisierungsgitters. Um das zu vermeiden, werden lediglich die Nicht-Nulleinträge der Matrix in jeder Zeile durchlaufen, sodass der Aufwand pro Komponente konstant wird. Das Vermeiden der Nulleinträge hat die Zeit für das Berechnen einer einzelnen Matrix-Vektor-Multiplikation von Sekunden auf Millisekunden reduziert.

```
1  for (int i = 0; i < dim; i++) {
2    double sum = 0.0;
3
4    /* Durchlaufe nur alle von Null verschiedenen Eintraege in der
5       aktuellen Zeile. */
6    for (int k = 0; k < 5; k++) {
7      Entry *currEntry = rowAcc[i][k];
8      if (currEntry != 0) {
9        sum += currEntry->val * vector[currEntry->col];
10     }
11   }
12
13   /* Eintrag berechnet. */
14   temp[i] = sum;
15 }
```

Listing 3.2: Matrix-Vektor-Multiplikation

Vorwärts-/Rückwärtseinsetzen Das Vorwärts-/Rückwärtseinsetzen für $A \cdot x = b$ ist ähnlich elementar wie die Matrix-Vektor-Multiplikation, da beides bei jedem Schritt der Iteration durchgeführt werden muss. Bei dem Vorwärtseinsetzen lassen sich die Komponenten des Ergebnisvektors x durch

$$x_k = \frac{1}{a_{kk}} \left(b_k - \sum_{j=1}^{k-1} a_{kj} x_j \right), \; x = 1, \ldots, n \tag{3.3}$$

bestimmen, bei dem Rückwärtseinsetzen hingegen durch

$$x_k = \frac{1}{a_{kk}} \left(b_k - \sum_{j=k+1}^{n} a_{kj} x_j \right), \; x = n, \ldots, 1 \tag{3.4}$$

Erneut werden bei diesem Verfahren die Nulleinträge der Matrix vermieden, um auch hier die Laufzeit im Bereich $\mathcal{O}(n^2)$ zu halten. Zu Beachten ist, dass in diesem Fall die Matrix sowohl die L- als auch die R-Matrix beinhaltet.

```
1   /* Vorwaertseinsetzen. */
2   double *r = new double[dim];
3
4   for (int i = 1; i <= dim; i++) {
5     double sum = 0.0;
6
7     for (int d = 0; d < 5; d++) {
8       Entry *currEntry = rowAcc[i - 1][d];
9       if (currEntry != 0 && currEntry->col >= 0
10          && currEntry->col < i - 1) {
11        sum += currEntry->val * r[currEntry->col];
12      }
13    }
14
15    /* Da die Eintraege auf der Hauptdiagonale von L alle eins sind,
16       muessen wir 1/a_11 nicht noch multiplizieren. */
17    r[i - 1] = y[i - 1] - sum;
18  }
19
20  /* Rueckwaertseinsetzen */
21  double *z = new double[dim];
22  for (int i = dim; i >= 1; i--) {
23    double sum = 0.0;
24
25    for (int d = 0; d < 5; d++) {
26      Entry *currEntry = rowAcc[i - 1][d];
27      if (currEntry != 0 && currEntry->col < dim
28          && currEntry->col >= i) {
29        sum += currEntry->val * z[currEntry->col];
30      }
31    }
32
33    z[i - 1] = 1.0 / getValue(i - 1, i - 1)
```

```
34                  * (r[i - 1] - sum);
35    }
```

Listing 3.3: Vorwärts-/Rückwärtseinsetzen

ILU-Zerlegung Der Algorithmus für die ILU-Zerlegung wurde weitestgehend direkt übernommen, wobei allerdings auch hier nur die Nicht-Nulleinträge beachtet werden, um eine möglichst hohe Ausführungsgeschwindigkeit zu erreichen. Weiterhin lässt sich der hier verwendeten Implementierung auch der Grund für die doppelte Indizierung der Matrix entnehmen. Während es für die vorherigen Algorithmen ausreichen würde, die Nicht-Nulleinträge einer jeden Zeile der Matrix zu erhalten, wird bei der ILU-Zerlegung die Matrix sowohl zeilen- als auch spaltenweise durchlaufen.

```
1   for (int k = 1; k <= size - 1; k++) {
2     for (int d2 = 0; d2 < 5; d2++) {
3       Entry *sEntry = matrix->colAcc[k - 1][d2];
4       if (sEntry != 0) {
5         int i = sEntry->row + 1;
6         if (i > size || i < k + 1) {
7           continue;
8         }
9         if (sEntry != 0 && sEntry->val != 0.0) {
10          sEntry->val /= matrix->getValue(k - 1, k - 1);
11
12          for (int d = 0; d < 5; d++) {
13            Entry *currEntry = matrix->rowAcc[i - 1][d];
14            if (currEntry != 0
15                && currEntry->col < size
16                && currEntry->col >= k) {
17              j = currEntry->col + 1;
18              currEntry->val -= sEntry->val
19                                * matrix->getValue(k - 1,
20                                                   j - 1);
21            }
22          }
23        }
24      }
25    }
26  }
```

Listing 3.4: ILU-Zerlegung

Es sei an dieser Stelle noch einmal darauf hingewiesen, dass die ILU-Zerlegung stark sequentiell abläuft und daher nur schwer parallelisierbar ist.

Richardson-Iteration Die tatsächliche Implementierung der Richardson-Iteration lässt sich folgendem Programmcode entnehmen.

```
1     /* x = x - (w(LU)^-1 x - w(LU)^-1 b) */
2     double *temp = copy(xn, size);
3     m->multiply(temp);
4     ILUM->multiplyWithInverseAsILU(temp);
5     Matrix::multiply(omega, temp, size);
6     Matrix::sub(temp, btilde, size);
7
8     /* Bevor wir den finalen Schritt in der Iteration machen, pruefen wir,
9        ob wir die notwendige Genauigkeit bereits erreicht haben (Dies ist
10       wegen der asynchronen Berechnung der Norm des Vektors
11       notwendig). */
12    if (!firstRun) {
13      norm = futureNorm.result();
14      /* Falls das Resultat genau genug ist, verlassen wir die Funktion
15         und rechnen nicht weiter. */
16      if (norm < e) {
17        delete[] temp;
18        break;
19      }
20    } else {
21      firstRun = false;
22    }
23
24    Matrix::sub(xn, temp, size);
25    delete[] temp;
26
27    /* Berechne die Genauigkeit asynchron. */
28    temp = copy(xn, size);
29    futureNorm = QtConcurrent::run(calcNorm, temp, m, b, size);
30
31    iterCount++;
32
33    /* Informiere den Benutzer ueber die aktuelle Genauigkeit und die
34       aktuelle Anzahl von Iterationsschritten. */
35    emit newAccuracy(QString::number(norm));
36    emit newIterationSteps(QString::number(iterCount));
37
38
39    /* Falls wir pausiert wurden, halte hier an. */
40    while (paused) {
41      mutex.lock();
42      wcondition.wait(&mutex);
43      mutex.unlock();
44    }
45  } while (!stopped);
```

Listing 3.5: ILU-Zerlegung

Die eigentliche Iteration findet in den Zeilen 3 bis 7 statt, ehe in den Zeilen 13 bis 23 das Ergebnis der erwähnten asynchronen Berechnung des Residuums abgefragt wird. Ist

dieses noch nicht genau genug, so wird in Zeile 24 die letzte Berechnung des Iterationsschritts vorgenommen. In Zeile 29 beginnt die erneute asynchrone Berechnung des Residuums.

Auch sieht man hier, dass die meisten Berechnungsfunktionen keinen neuen Vektor o.ä. zurückliefern, sondern stattdessen destruktive Updates auf den bereits existenten Datenobjekten ausführen (Dies lässt sich auch den vorher beschriebenen Algorithmen entnehmen). Dies bringt zwar für schwerer lesbaren Code mit sich, sorgt jedoch für eine deutlich höhere Ausführungsgeschwindigkeit, da auf die ständige Erzeugung neuer Objekte (und insbesondere der Ressourcenallokation) soweit wie möglich verzichtet wird.

Dem Quellcode im Anhang dieser Arbeit lässt sich zudem entnehmen, dass die gesamte Richardson-Iteration in einen eigenen Thread ausgelagert wurde, insbesondere damit die GUI reaktiv bleibt und so auch die Steuerung der Berechnung weiterhin möglich ist.

4. Empirische Untersuchung

Um den Aufwand der Richardson-Iteration mit ILU-Vorkonditionierung angewendet auf die Konvektions-Diffusionsgleichung empirisch zu bestimmen, wird das Programm mit drei verschiedenen Konstellationen durchlaufen. Dabei werden nacheinander

- die Gitterweite h bzw. die Gitteranzahl n,
- die Schranke e der Defektkorrektur

und

- der Faktor des Diffusionsterms

variiert, während die anderen Parameter konstant gehalten werden. Mithilfe dieses Aufbaus sollen sich die Einflüsse der verschiedenen Parameter auf die Richardson-Iteration bestimmen lassen. Um weiterhin die Ausführungsgeschwindigkeit objektiv beurteilen zu können und Messungenauigkeiten (beispielsweise durch Prozess-Scheduling) zu unterdrücken, wird statt der realen CPU-Zeit lediglich die Anzahl der Iterationen angegeben, die während der Richardson-Iteration tatsächlich durchlaufen werden. Die exakten Messdaten zu den einzelnen Auswertungen lassen sich dem Anhang entnehmen.

1. Auswertung

Für die erste Auswertung werden drei verschiedene Konstellationen betrachtet. Zum einen die Poisson-Gleichung ohne zusätzlichen Konvektionsterm

$$-\Delta u \equiv 1 \text{ in } \Omega \tag{4.1}$$

wobei $n = 100$ konstant gehalten wird. Zum anderen die Konvektions-Diffusionsgleichung

$$-0.1 \cdot \Delta u + \begin{pmatrix} 0.5 \\ -1.5 \end{pmatrix} \cdot \nabla u \equiv 1 \text{ in } \Omega \tag{4.2}$$

ebenfalls mit $n = 100$ und als dritte Konstellation mit $n = 250$ die Gleichung

$$-0.01 \cdot \Delta u + \begin{pmatrix} 2.0 \\ -1.0 \end{pmatrix} \cdot \nabla u \equiv 1 \text{ in } \Omega. \tag{4.3}$$

Für alle drei Gleichungen wird die Schranke e mit $e = \frac{1}{2^k}$, $3 \leq k \leq 14$ variiert. Die Visualisierung der Messdaten befindet sich in Abbildung 4.1, wobei eine einfache Logarithmierung der x-Koordinatenachse vorgenommen wurde.

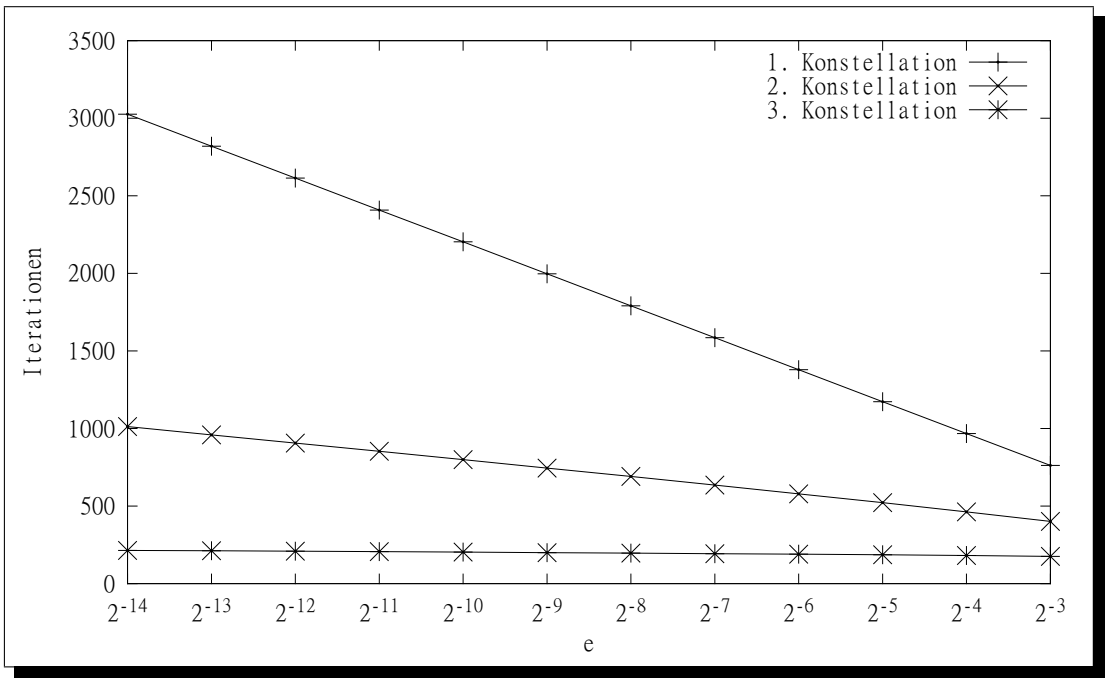


Abbildung 4.1.: Messergebnisse der ersten Auswertung mit zur Basis Zwei logarithmierter x-Achse.

Die Messpunkte von allen drei Gleichungen lassen sich jeweils zu Geraden unterschiedlicher Steigung verbinden, was den Rückschluss auf logarithmisches Verhalten in Abhängigkeit von e^{-1} zulässt.

$$\Rightarrow \text{Anzahl Iterationen} \in O\left(\log\left(\frac{1}{e}\right)\right)$$

Insbesondere bedeutet dies an dieser Stelle eine lineare Konvergenzrate des Verfahrens.

2. Auswertung

Für die zweite Auswertung werden erneut die drei Gleichungen

$$-\Delta u \equiv 1 \text{ in } \Omega \tag{4.4}$$

$$-0.1 \cdot \Delta u + \begin{pmatrix} 0.5 \\ -1.5 \end{pmatrix} \cdot \nabla u \text{ in } \Omega \tag{4.5}$$

$$-0.05 \cdot \Delta u + \begin{pmatrix} 2.0 \\ -1.0 \end{pmatrix} \cdot \nabla u \text{ in } \Omega \tag{4.6}$$

aus der ersten Auswertung betrachtet (die dritte Gleichung wurde aus Gründen der Konvergenz geringfügig modifiziert), wobei hier eine konstante Fehlerschranke von $e = 10^{-4}$ zugrunde gelegt wird. Die Auflösung des Gitters wird mit $n = 2^k$, $2 \leq k \leq 9$ variiert. Abbildung 4.2 zeigt die graphische Darstellung der Messergebnisse.

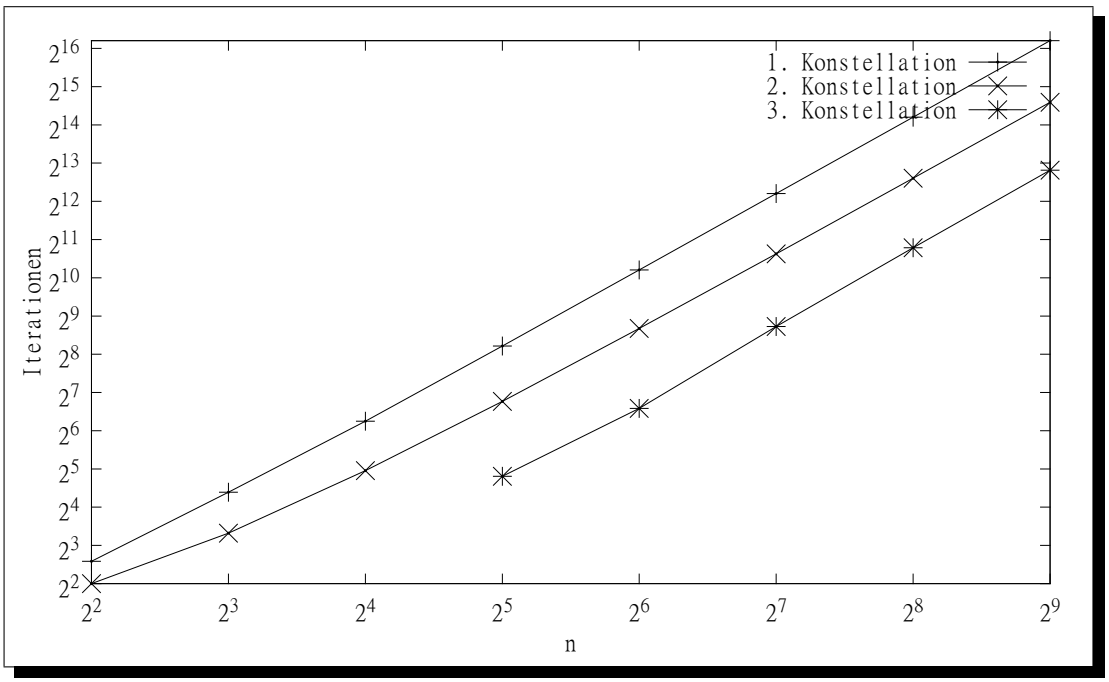


Abbildung 4.2.: Messergebnisse der zweiten Auswertung. Beide Achsen wurden zur Basis Zwei logarithmiert.

Obwohl die Messdaten sich bei den konvektionsdominanten Gleichungen zunächst stabilisieren müssen, erkennt man deutlich, dass es sich um Geraden handelt, die zudem ähnliche Steigungen aufweisen. Aufgrund der doppelten Logarithmierung lässt dies den Schluss zu, dass den Werten ein polynomiales Verhalten zugrunde liegt.

Werden die Messpunkte $(8 \mid 21)$ sowie $(256 \mid 18855)$ im doppelt logarithmierten Koordinatensystem miteinander verbunden, so besitzt die Gerade der ersten Konstellation eine Steigung von etwa 1.962. Die zweite Gerade, die durch die Messpunkte $(8 \mid 21)$ und $(256 \mid 6222)$ definiert wird, hat hingegen eine Steigung von etwa 1.642. Für die dritte Konstellation lassen sich die Messpunkte $(64 \mid 96)$ sowie $(512 \mid 7196)$ miteinander verbinden, was eine Steigung von etwa 2.076 ergibt.

Insgesamt bedeutet das, dass der Aufwand für die Richardson-Iteration etwa quadratisch mit der Auflösung des Diskretisierungsgitters ansteigt. Dies entspricht auch genau der theoretischen Erwartung aufgrund $cond\,(A) \sim n^2$.

$$\Rightarrow \text{Anzahl Iterationen} \in O\left(n^2\right) = O\left(h^{-2}\right)$$

3. Auswertung

Bei der dritten Auswertung wird die Konvektions-Diffusionsgleichung

$$-\epsilon \cdot \Delta u + \begin{pmatrix} 0.5 \\ -1.5 \end{pmatrix} \cdot \nabla u \equiv 1 \tag{4.7}$$

mit $n = 250$ und $e = 10^{-4}$ betrachtet. Der Diffusionsparameter ϵ wird durch 10^{-k} mit $0 \leq k \leq 5$ variiert. Die Messwerte werden in Abbildung 4.3 unter Zuhilfenahme einer einfachen Logarithmierung dargestellt.

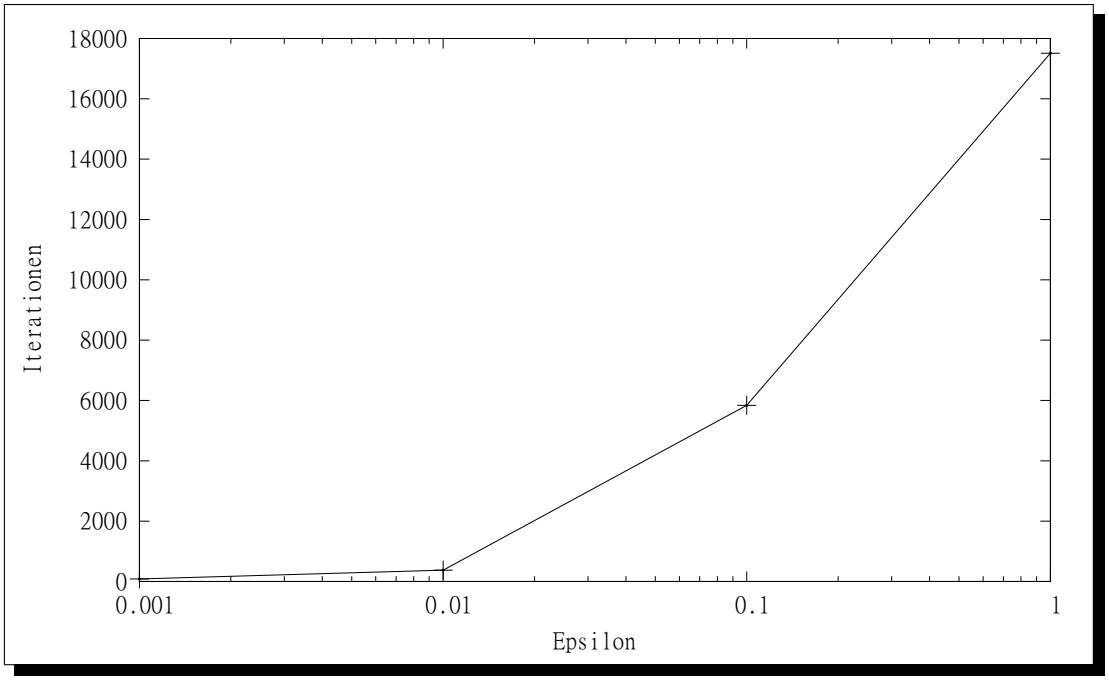


Abbildung 4.3.: Messergebnisse der dritten Auswertung mit zur Basis Zehn logarithmierten x-Achse.

Wie man erkennen kann, sind die Messergebnisse nicht ganz eindeutig, womit sich der Aufwand in Abhängigkeit des Diffusionsparameters nicht ohne weiteres bestimmen lässt. Aus diesem Grund wird an dieser Stelle das Residuum (d.h. $\|b - A \cdot x\|_\infty$) in Abhängigkeit der Iterationen geplottet, wobei für jedes ϵ ein neuer Datensatz geplottet wird. Unter Zuhilfenahme einer einfachen Logarithmierung der y-Achse ergibt sich das Diagramm in Abbildung 4.4.

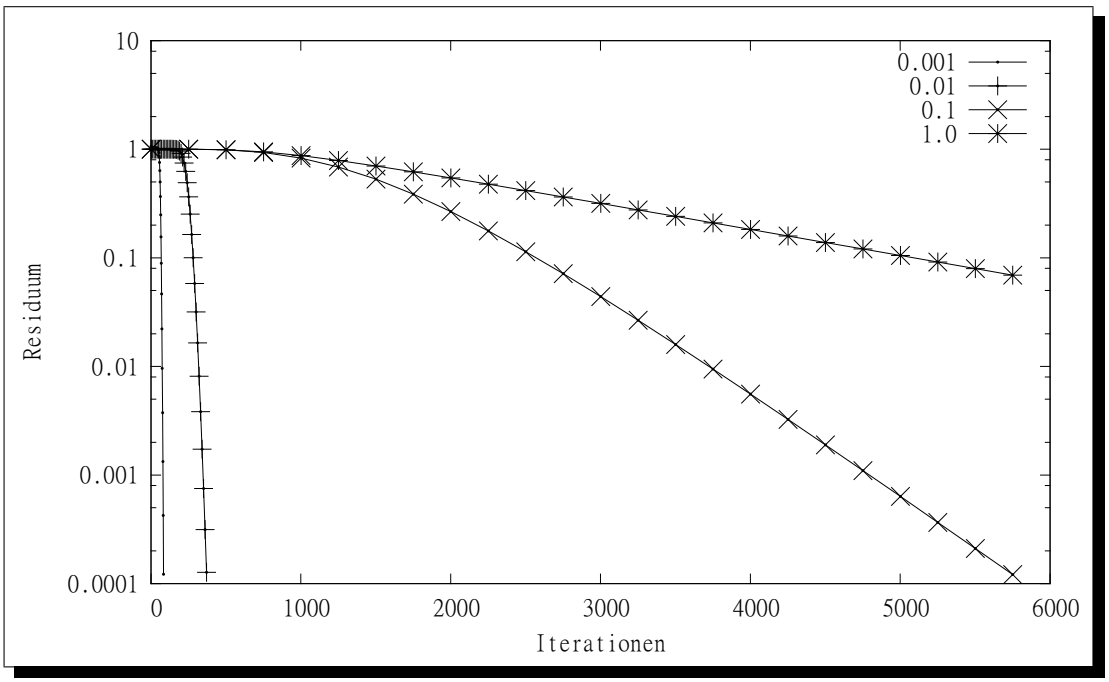


Abbildung 4.4.: Messergebnisse der dritten Auswertung. Die y-Achse wurde zur Basis Zehn logarithmiert.

Nun erkennt man, dass im stark konvektionsdominanten Fall das Verfahren schnell konvergiert, wenn der Diffusionsterm entsprechend gering ist. Wird der Diffusionsterm stärker ($\epsilon = 0.1$, bzw. $\epsilon = 1.0$) oder befindet sich dieser sogar in der Größenordnung des Konvektionsterms, so konvergiert das Verfahren nur langsam.

5. Ausblick

Nachdem in den vorherigen Kapiteln die Schwächen der verwendeten Algorithmen gezeigt wurden, beschäftigt sich dieses Kapitel damit, wie sich das gegebene Verfahren beschleunigen, die ILU-Zerlegung sich abwandeln lässt und wie man das gesamte Diskretisierungsverfahren numerisch stabilisieren kann. Es sei angemerkt, dass sowohl die beiden Verfahren in 5.1 als auch das Upwinding aus 5.3 im C++-Programm implementiert wurden.

5.1. Konvergenzbeschleunigung

5.1.1. Gedämpftes Richardson-Verfahren

Eine Dämpfung der Richardson-Iteration erreicht man durch das Anfügen eines reellen Dämpfungsparameters ω, mit dem man das Residuum gewichtet [Pol02, Hac93]. Das Verfahren wird entsprechend über- oder unterrelaxiert. Aus der ursprünglichen Form

$$x_{n+1} = x_n - (Ax_n - b) \tag{5.1}$$

wird dann das gedämpfte oder auch relaxierte Richardson-Verfahren

$$x_{n+1} = x_n - \omega \cdot (Ax_n - b) = x_n - (\omega \cdot Ax_n - \omega \cdot b)\,. \tag{5.2}$$

Durch Wahl von $\omega = 1.0$ erhält man das ursprüngliche, stationäre Richardson-Verfahren. Die Dämpfung lässt sich einsetzen, um die Konvergenz nochmals leicht zu beschleunigen, wie sich Abbildung 5.1 entnehmen lässt. Hierbei wird die Gleichung

$$-0.1 \cdot \Delta u + \begin{pmatrix} -1.0 \\ 1.0 \end{pmatrix} \cdot \nabla u \equiv 1 \tag{5.3}$$

bei $n = 250$, $e = 10^{-4}$ betrachtet.

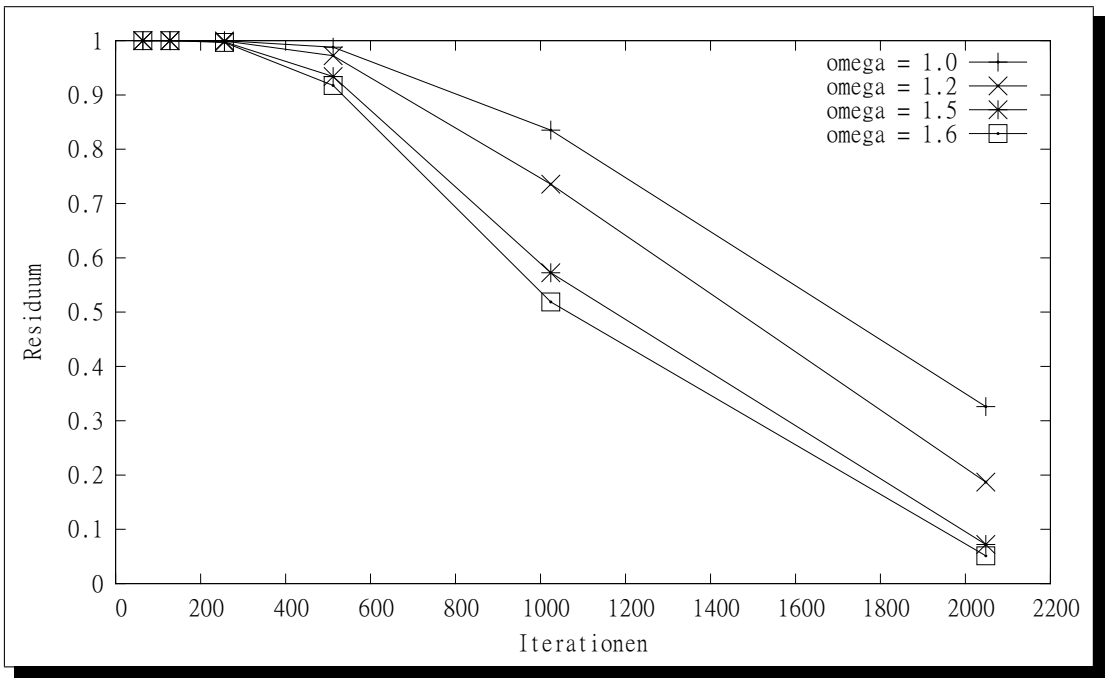


Abbildung 5.1.: Das Residuum in Abhängigkeit der Iterationsschritte bei verschiedenen Werten für ω.

Zu erkennen ist, dass eine leichte Überrelaxierung bereits für eine verbesserte Konvergenz sorgen kann. Ist ω jedoch zu groß, so ist sogar Divergenz zu erwarten.

5.1.2. Über-/Unterrelaxation der Hauptdiagonalen

Für eine zweite Möglichkeit zur Beschleunigung des Verfahrens wird auf Vorschlag des Betreuers hin eine leichte Über- bzw. Unterrelaxation der Hauptdiagonaleinträge untersucht. Hierzu werden die Einträge a_{ii} der ILU-Zerlegung modifiziert:

$$a_{ii} := a_{ii} + \underbrace{\gamma}_{\in\mathbb{R}} \cdot \sum_{i \neq j} |a_{ij}| \tag{5.4}$$

Durch die Wahl von $\gamma = 0.0$ erhält man auch hier das ursprüngliche, unmodifizierte Verfahren. Zur Untersuchung wird obige Gleichung

$$-0.1 \cdot \Delta u + \begin{pmatrix} -1.0 \\ 1.0 \end{pmatrix} \cdot \nabla u \equiv 1 \tag{5.5}$$

bei $n = 250$, $e = 10^{-4}$ betrachtet.

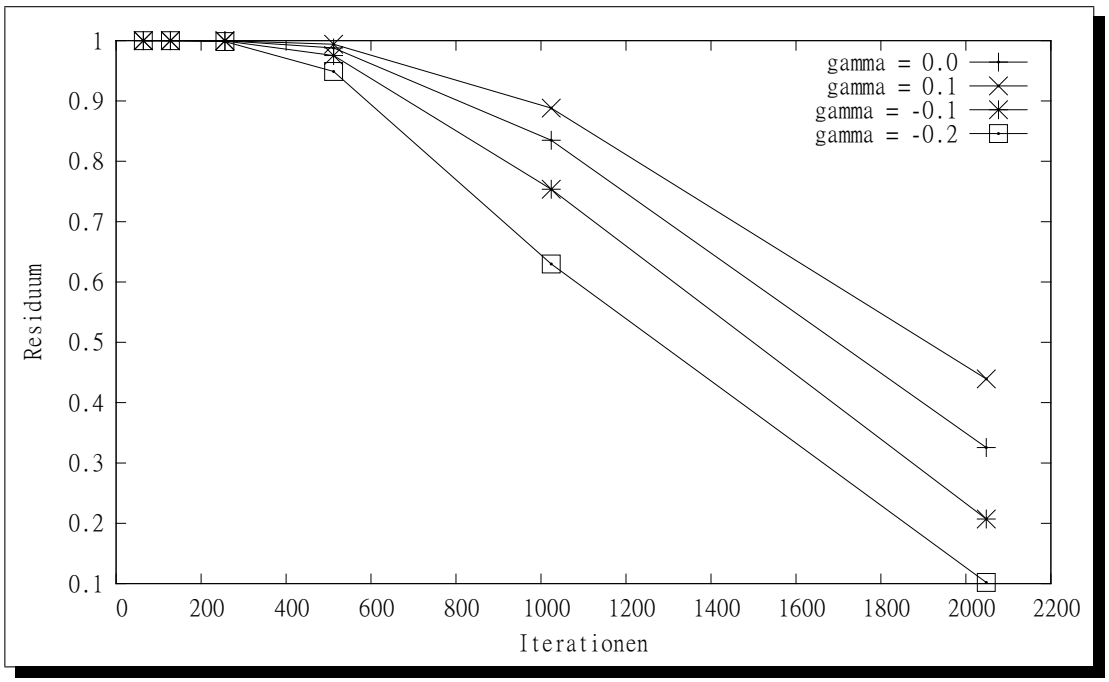


Abbildung 5.2.: Das Residuum in Abhängigkeit der Iterationsschritte bei verschiedenen Werten für γ.

Die Über-/Unterrelaxation kann das Verfahren also nun nochmals beschleunigen.

5.2. Modifikationen der ILU-Zerlegung

In diesem Abschnitt der Arbeit wird beschrieben, wie sich das in Abschnitt 2.4 aufgeführte Verfahren zur unvollständigen Zerlegung einer Matrix modifizieren oder erweitern lässt. Die erweiterten Verfahren wurden zumeist in Hinsicht auf Geschwindigkeitsgewinn oder numerische Stabilität entwickelt oder aber damit die Zerlegung einen besseren Präkonditionierer darstellt. Die Verfahren werden lediglich kurz erläutert, jedoch nicht ausführlich behandelt. Für genauere Ausführungen sei auf weiterführende Literatur verwiesen.

ILU(p)-Zerlegung Bei der ursprünglichen ILU-Zerlegung erhält die resultierende Matrix eine identische Besetzungsstruktur wie die Ursprungsmatrix. Die ILU(p)-Zerlegung verzichtet auf diese zusätzliche Bedingung und bringt - je nach Wahl des sogenannten Fill-Levels p - zusätzliche Einträge in die Matrizen L und U. Ein Fill-Level von $p = 0$ entspricht dabei dem Originalverfahren.

Da die Matrix $(LU)^{-1}$ die Matrix A^{-1} besser approximiert, erlaubt dieser Vorkonditionierer zumeist eine deutlich schnellere Konvergenz. Die verbesserte Konvergenz muss jedoch mit erhöhtem Rechen- und Speicheraufwand für die Zerlegung bezahlt werden. Auch benötigt das Lösen der Gleichung

$$(LU)^{-1} x = y \tag{5.6}$$

deutlich mehr Aufwand [Sch11].

ILUT(p, ϵ)-Zerlegung Ein Problem, welches bei den obigen Verfahren auftritt, ist, dass sehr viel Rechenzeit für Einträge verschwendet wird, die ohnehin fast Null sind. Aus diesem Grunde werden bei dieser Zerlegung bestimmte Elemente, die unterhalb eines durch ϵ bestimmten Schwellwerts sind, ignoriert [Sch11].

IC-Zerlegung Die unvollständige Cholesky-Zerlegung lässt sich analog zur unvollständigen LU-Zerlegung durch

$$A = L \cdot L^T + R \tag{5.7}$$

beschreiben [Mei11]. Hierbei ist deutlich weniger Rechenaufwand und Speicherbedarf notwendig, da lediglich die Berechnung und Speicherung der unteren Dreiecksmatrix notwendig ist. Gegenüber der ursprünglichen ILU-Zerlegung ist dieses Verfahren insbesondere bei symmetrischen Matrizen numerisch stabiler [Beb11].

5.3. Upwinding

Eine Möglichkeit, das Verfahren der Diskretisierung selbst zu stabilisieren besteht in der Verwendung von Upwinding-Diskretisierung [Bad01]. Statt wie in 2.1.2 die Ableitungen durch den zentralen Differenzenquotienten zu approximieren, werden bei dem Upwinding die einseitigen Differenzenquotienten verwendet und stets der Konvektion (d.h. insbesondere dem Datenfluss) entgegen gerichtet. Man setzt also

$$\frac{\partial u}{\partial x} \approx \begin{cases} \frac{u_{i+1,j}-u_{i,j}}{h} & \text{falls } \beta_1 < 0 \\ \frac{u_{i,j}-u_{i-1,j}}{h} & \text{falls } \beta_1 > 0 \end{cases} \tag{5.8}$$

beziehungsweise

$$\frac{\partial u}{\partial y} \approx \begin{cases} \frac{u_{i,j+1}-u_{i,j}}{h} & \text{falls } \beta_2 < 0 \\ \frac{u_{i,j}-u_{i,j-1}}{h} & \text{falls } \beta_2 > 0 \end{cases} \tag{5.9}$$

Durch die Verwendung der Upwinding-Diskretisierung ist nun die Forderung nach der Ungleichung

$$h < \frac{2 \cdot \epsilon}{\|\beta\|_1} \tag{5.10}$$

zwischen den Koeffizienten nicht länger notwendig. Tatsächlich führt die Verwendung dieser modifizierten Diskretisierung unabhängig von der Stärke und Richtung des Konvektionsterms dazu, dass die Systemmatrix immer die M-Matrix-Eigenschaft besitzt. Neben dem etwas größeren Aufwand zum Aufstellen der Systemmatrix (aufgrund der Fallunterscheidungen) ist vor allem die geringere Genauigkeit dieser Diskretisierung als Nachteil zu nennen. Während die zentralen Differenzenquotienten zu einem Fehler der Größenordnung $\mathcal{O}(h^2)$ führen, ist der Diskretisierungsfehler durch die Upwind-Diskretisierung nur in $\mathcal{O}(h)$ (vgl. [Bad01]).

Mithilfe des Upwindings kann nun auch beispielsweise die fehlerhafte Berechnung, wie sie in Abbildung 2.2 zu finden ist, korrigiert werden. Abbildung 5.3 stellt die korrekte Lösung dar.

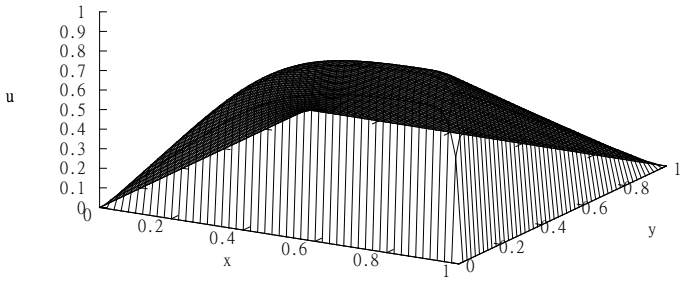


Abbildung 5.3.: Dreidimensionale Visualisierung der Konvektions-Diffusionsgleichung ($n = 50$, $\beta = (0.5, -1.0)^T$, $\epsilon = 0.005$, $e = 10^{-4}$, $f \equiv 1$).

6. Fazit

Im Laufe dieser Arbeit wurde die Richardson-Iteration als iteratives Verfahren mit der ILU-Zerlegung als Vorkonditionierer vorgestellt. Es stellte sich heraus, dass die ILU-Zerlegung grundsätzlich als Vorkonditionierer geeignet ist, da diese die Iteration zum einen stark beschleunigt und zum anderen in den meisten Fällen überhaupt erst die Konvergenz ermöglicht. Nachteile des Vorkonditionierer waren allerdings die relativ starken Forderungen an die Systemmatrix (M-Matrix-Eigenschaft) und die Tatsache, dass die Anzahl der Iterationsschritte trotz Vorkonditionierer noch immer quadratisch mit n anstiegen.

Der erste Nachteil ließ sich durch die Verwendung des Upwinding-Verfahrens kompensieren, da dadurch die notwendigen Eigenschaften unabhängig von den Koeffizienten immer garantiert werden konnten. Dies geschah allerdings auf Kosten der Genauigkeit.

Der zweite Nachteil ließ sich nur teilweise ausgleichen. Die Verwendung von Dämpfungsparametern für das Iterationsverfahren und für die Hauptdiagonaleinträge der Matrix senkten zwar die notwendige Schrittzahl im Verfahren, können jedoch das asymptotische Verhalten nicht verändern.

Als weiterführende Möglichkeit bliebe allerdings die Vorkonditionierung selbst zu verbessern. Dies wäre durch eine etwas vollständigere Zerlegung durchaus möglich - dies würde jedoch auf Kosten von Speicherbedarf und Aufwand geschehen.

A. Quellcode

```
1  #include <QtGui/QApplication>
2  #include "mainwindow.h"
3  #include <QWaitCondition>
4  /**
5   * Dieses Programm wurde im Rahmen der Bachelorarbeit "Die inkomplette LR-
6   * Zerlegung fuer Konvektions-Diffusionsgleichungen" entwickelt. Es stellt
7   * eine grundlegende, simple GUI bereit, mit der die Parameter fuer die
8   * Gleichungen eingegeben werden koennen. Die Gleichung kann dann
9   * diskretisiert und geloest werden, wobei dies mithilfe der inkompletten
10  * LR-Zerlegung als Vorkonditionierer und der Richardson-Iteration
11  * geschieht.<br>
12  * Das Programm selbst verwendet relativ wenig Fehlerbehandlung und
13  * wiederverwendbaren Code, sondern setzt stattdessen auf sehr spezifische
14  * Datentypen und Code, um Effizienz und Performance zu gewaehrleisten.
15  *
16  * @author Nils Christian Ehmke
17  */
18 int main(int argc, char *argv[]) {
19   QApplication a(argc, argv);
20
21   /* Erzeuge das Fenster und zeige es an. */
22   MainWindow w;
23   w.show();
24
25   return a.exec();
26 }
```

Listing A.1: main.cpp

```
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5  #include "workthread.h"
6
7  namespace Ui {
8    class MainWindow;
9  }
10
11 /**
12  * @brief Das Hauptfenster der Applikation.
13  *
14  * Dies ist das Hauptfenster der Anwendung. Dieses stellt dem Benutzer die
15  * Moeglichkeit bereit, die Parameter einzugeben und die Berechnung
```

```
16  * prinzipiell zu steuern.
17  *
18  * @author Nils Christian Ehmke
19  */
20 class MainWindow : public QMainWindow {
21   Q_OBJECT
22
23 public:
24   /**
25    * Erzeugt eine neue Instanz dieser Klasse.
26    *
27    * @param parent
28    *                 Der Parent dieses Fensters.
29    */
30   explicit MainWindow(QWidget *parent = 0);
31
32   /**
33    * Der Destruktor fuer diese Klasse.
34    */
35   ~MainWindow();
36
37   /**
38    * Das Event, welches ausgeloest wird, wenn das Fenster vergroessert
39    * oder verkleinert wird. Es ist lediglich zum ueberschreiben hier
40    * aufgefuehrt.
41    */
42   void resizeEvent(QResizeEvent *event);
43
44
45 private:
46   /**
47    * Dieses Feld beinhaltet die GUI-Komponenten.
48    */
49   Ui::MainWindow *ui;
50
51   /**
52    * Dieses Feld ist fuer den Thread, der die Berechnung durchfuehrt.
53    */
54   Workthread *workthread;
55   /**
56    * Die" Leinwand", auf der spaeter die Visualisierung gezeichnet wird.
57    */
58
59   QPixmap *map;
60   /**
61    * Die Szene, in welchem die obige "Leinwand" gespeichert wird.
62    */
63   QGraphicsScene *scene;
64
65 private slots:
66   /**
67    * Dieser Slot signalisiert, dass die Applikation zu beenden ist.
```

```
68      */
69     void on_actionBeenden_triggered();
70
71     /**
72      * Dieser Slot signalisiert, dass das Informations-Fenster anzuzeigen
73      * ist.
74      */
75     void on_actionUeber_triggered();
76
77     /**
78      * Dieser Slot signalisiert, dass der Datei-Dialog zum Speichern der
79      * Visualisierung angezeigt werden soll.
80      */
81     void on_btnFindFile_clicked();
82
83     /**
84      * Dieser Slot signalisiert, dass die Checkbox fuer die dreidimensionale
85      * Visualisierung (de)selektiert wurde.
86      */
87     void on_chkBoxPlot3D_toggled(bool checked);
88
89     /**
90      * Dieser Slot signalisiert, dass die Berechnung zu pausieren /
91      * fortzufuehren ist.
92      */
93     void on_btnPauseResumeCalculation_clicked();
94
95     /**
96      * Dieser Slot signalisiert, dass die Berechnung vorzeitig zu stoppen
97      * ist.
98      */
99     void on_btnStopCalculation_clicked();
100
101    /**
102     * Dieser Slot signalisiert, dass der Parameter "n" geaendert wurde.
103     */
104    void on_leN_textChanged(QString );
105
106    /**
107     * Dieser Slot signalisiert, dass die Berechnung mit den aktuellen
108     * Parametern zu starten ist.
109     */
110    void on_btnStartCalculation_clicked();
111
112    /**
113     * Dieser Slot ermoeglicht es, dass der Text fuer die aktuelle
114     * "Genauigkeit" neu gesetzt wird.
115     *
116     * @param str
117     *              Der neue Text fuer die "Genauigkeit".
118     */
119    void setAccuracyText(QString str);
```

```
120
121     /**
122      * Dieser Slot ermoeglicht es, dass der Test fuer die aktuelle Anzahl
123      * von Iterationsschritten neu gesetzt wird.
124      *
125      * @param str
126      *              Der neue Text fuer die Iterationsschritte.
127      */
128     void setIterationStepsText(QString str);
129
130     /**
131      * Mit diesem Slot kann eine beliebige Information im Hauptfenster
132      * "geloggt werden", d.h. die Nachricht wird mit Zeitstempel in der
133      * Logkomponente angezeigt.
134      *
135      * @param str
136      *              Der zu loggende Text.
137      */
138     void logInformation(QString str);
139
140     /**
141      * Mit diesem Slot kann eine Visualisierung angezeigt werden.
142      *
143      * @param img
144      *               Die zu zeichnende Visualisierung.
145      */
146     void set2DGraphic(QImage img);
147
148     /**
149      * Mit diesem Slot koennen die Buttons, die Anzeige etc. nach Beendigung
150      * oder Abbruch der Berechnung zuruechgesetzt werden.
151      */
152     void resetEverything();
153 };
154
155 #endif // MAINWINDOW_H
```

Listing A.2: mainwindow.h

```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3  #include "matrix.h"
4  #include "aboutdialog.h"
5  #include "QTime"
6  #include <QProcess>
7  #include <QMessageBox>
8  #include <QFileDialog>
9
10 MainWindow::MainWindow(QWidget *parent) :
11     QMainWindow(parent), ui(new Ui::MainWindow) {
12     ui->setupUi(this);
13     this->setWindowState(Qt::WindowMaximized);
14     /* Setze alle Pointer auf 0, damit sie einen definierten Wert haben. */
```

```
15    workthread = 0;
16    map = 0;
17    scene = 0;
18  }
19
20  MainWindow::~MainWindow() {
21    delete ui;
22  }
23
24  void MainWindow::logInformation(QString str) {
25    /* Setze den Nachrichtenstring mit Zeitstempel zusammen und zeige ihn
26       an. */
27    str = QTime::currentTime().toString() + ": " + str;
28    ui->plTextEditLog->appendPlainText(str);
29  }
30
31  void MainWindow::setAccuracyText(QString str) {
32    str = "Aktuelle Genauigkeit: " + str;
33    ui->lblCurrAcc->setText(str);
34  }
35
36  void MainWindow::setIterationStepsText(QString str) {
37    str = "Aktuelle Anzahl Iterationsschritte: " + str;
38    ui->lblCurrIter->setText(str);
39  }
40
41  void MainWindow::resetEverything() {
42    /* Setze die Buttons wieder zurueck. */
43    this->ui->btnStartCalculation->setEnabled(true);
44    this->ui->btnStopCalculation->setEnabled(false);
45    this->ui->btnPauseResumeCalculation->setEnabled(false);
46
47    /* Es werden keine weiteren Genauigkeiten oder Iterationsschritte
48       angezeigt. */
49    this->setAccuracyText("n/a");
50    this->setIterationStepsText("n/a");
51  }
52
53  void MainWindow::on_btnStartCalculation_clicked() {
54    /* Blockiere den Start-Button und aktiviere den Stop-Button. */
55    this->ui->btnStartCalculation->setEnabled(false);
56    this->ui->btnStopCalculation->setEnabled(true);
57    this->ui->btnPauseResumeCalculation->setEnabled(true);
58
59    /* Hole und konvertiere die Parameter. */
60    int n = ui->leN->text().toInt();
61    double beta1 = ui->leBeta1->text().toDouble();
62    double beta2 = ui->leBeta2->text().toDouble();
63    double epsilon = ui->leEpsilon->text().toDouble();
64    double omega = ui->leOmega->text().toDouble();
65    double gamma = ui->leGamma->text().toDouble();
66    double e = ui->leE->text().toDouble();
```

```
67    bool useUpwinding = ui->chkBoxUseUpwinding->isChecked();
68    QString fileName3DPlot("");
69    if (ui->chkBoxPlot3D->isChecked()) {
70      fileName3DPlot = ui->le3DPlotFileName->text();
71    }
72
73    /* Loesche die alten Anzeigeobjekte, um ein Speicherleck zu
74       vermeiden. */
75    if (map != 0) {
76      delete map;
77    }
78    if (scene != 0) {
79      delete scene;
80    }
81
82    /* Erzeuge die Objekte zur Anzeige neu. */
83    map = new QPixmap((n+1), (n+1));
84    scene = new QGraphicsScene();
85    scene->addPixmap(*map);
86
87    /* Leere die Leinwand, um zurueckgebliebenen Datenmuell zu vermeiden. */
88    ui->graphicsView->setScene(scene);
89    QImage img = map->toImage();
90    img.fill(16777215);
91    map->convertFromImage(img);
92    scene->update();
93
94    /* Stell sicher, dass die Leinwand auf die Groesse des Bildschirms
95       angepasst wird. */
96    ui->graphicsView->fitInView(scene->sceneRect(), Qt::KeepAspectRatio);
97
98    /* Bereite den neuen Arbeitsthread vor. */
99    this->workthread = new Workthread(epsilon, beta1, beta2, e, n, omega,
100                                      gamma, fileName3DPlot, useUpwinding);
101
102   /* Verbinde den neuen Arbeitsthread mit unseren Slots, damit er alles
103      anzeigen kann und wir auch mitbekommen, sobald er fertig ist. */
104   connect(this->workthread, SIGNAL(finished()), scene, SLOT(update()));
105   connect(this->workthread, SIGNAL(finished()), this,
106           SLOT(resetEverything()));
107   connect(this->workthread, SIGNAL(newAccuracy(QString)), this,
108           SLOT(setAccuracyText(QString)));
109   connect(this->workthread, SIGNAL(newIterationSteps(QString)), this,
110           SLOT(setIterationStepsText(QString)));
111   connect(this->workthread, SIGNAL(newLogMessage(QString)), this,
112           SLOT(logInformation(QString)));
113   connect(this->workthread, SIGNAL(new2DGraphic(QImage)), this,
114           SLOT(set2DGraphic(QImage)));
115
116   /* Starte den Arbeitsthread asynchron. */
117   this->workthread->start();
118 }
```

```
119
120 void MainWindow::on_btnStopCalculation_clicked() {
121   if (this->workthread != 0) {
122     /* Sofern ein Thread existiert, sende ihm das Zeichen zu stoppen und
123        warte auf ihn. */
124     this->workthread->stop();
125     this->workthread->wait();
126
127     /* Loesche den Thread. */
128     delete (this->workthread);
129     this->workthread = 0;
130   }
131 }
132
133 void MainWindow::on_btnPauseResumeCalculation_clicked() {
134   if (this->workthread != 0) {
135     /* Sofern ein Thread existiert, pausiere ihn bzw. lasse ihn
136        fortfahren. */
137     if (this->workthread->isPaused()) {
138       this->workthread->resume();
139     } else {
140       this->workthread->pause();
141     }
142   }
143 }
144
145 void MainWindow::on_actionBeenden_triggered() {
146   on_btnStopCalculation_clicked();
147   this->close();
148 }
149
150 void MainWindow::resizeEvent(QResizeEvent *event) {
151   if (scene != 0) {
152     ui->graphicsView->fitInView(scene->sceneRect(), Qt::KeepAspectRatio);
153   }
154   QWidget::resizeEvent(event);
155 }
156
157 void MainWindow::on_chkBoxPlot3D_toggled(bool checked) {
158   ui->le3DPlotFileName->setEnabled(checked);
159   ui->btnFindFile->setEnabled(checked);
160 }
161
162 void MainWindow::on_leN_textChanged(QString ) {
163   /* Hole den Parameter n. */
164   int n = ui->leN->text().toInt();
165   /* Berechne entsprechend h und zeige den Parameter an. */
166   double h = 1.0 / n;
167   ui->leH->setText(QString::number(h));
168 }
169
170 void MainWindow::on_btnFindFile_clicked() {
```

```
171   /* Bereite das Dialogfenster vor. Der User soll die Datei zum Speichern
172      finden können, akzeptiert werden nur pdf-Dateien. */
173   QFileDialog dialog(this);
174   dialog.setFileMode(QFileDialog::AnyFile);
175   dialog.setNameFilter("*.pdf");
176   dialog.setAcceptMode(QFileDialog::AcceptSave);
177
178   /* Zeig den Dialog an und hole ggf. den Dateinamen. */
179   if (dialog.exec()) {
180     ui->le3DPlotFileName->setText(dialog.selectedFiles().first());
181   }
182 }
183
184 void MainWindow::set2DGraphic(QImage img) {
185   map->convertFromImage(img);
186 }
187
188 void MainWindow::on_actionUeber_triggered() {
189   /* Erzeuge und zeige das About-Fenster an. */
190   AboutDialog dialog(this);
191   dialog.exec();
192 }
```

Listing A.3: mainwindow.cpp

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ui version="4.0">
3   <class>MainWindow</class>
4   <widget class="QMainWindow" name="MainWindow">
5    <property name="geometry">
6     <rect>
7      <x>0</x>
8      <y>0</y>
9      <width>761</width>
10     <height>936</height>
11    </rect>
12   </property>
13   <property name="windowTitle">
14    <string>Konvektions-Diffusionsgleichungen</string>
15   </property>
16   <property name="windowIcon">
17    <iconset resource="Ressourcen.qrc">
18     <normaloff>:/res/My-computer.ico</normaloff>:/res/My-computer.ico</
          iconset>
19   </property>
20   <widget class="QWidget" name="centralWdgt">
21    <layout class="QGridLayout" name="gridLayout" columnstretch="1,2">
22     <item row="0" column="0">
23      <widget class="QWidget" name="wdgt1" native="true">
24       <layout class="QGridLayout" name="gridLayout_3">
25        <property name="bottomMargin">
26         <number>0</number>
27        </property>
```

```
28          <item row="4" column="0">
29           <widget class="QWidget" name="wdgt2" native="true">
30            <layout class="QGridLayout" name="gridLayout_2">
31             <property name="margin">
32              <number>0</number>
33             </property>
34             <item row="1" column="0">
35              <widget class="QPushButton" name="btnStartCalculation">
36               <property name="text">
37                <string> Berechnung starten </string>
38               </property>
39              </widget>
40             </item>
41             <item row="1" column="2">
42              <widget class="QPushButton" name="btnStopCalculation">
43               <property name="enabled">
44                <bool>false</bool>
45               </property>
46               <property name="text">
47                <string>Berechnung stoppen</string>
48               </property>
49              </widget>
50             </item>
51             <item row="1" column="1">
52              <widget class="QPushButton" name="btnPauseResumeCalculation">
53               <property name="enabled">
54                <bool>false</bool>
55               </property>
56               <property name="text">
57                <string>Berechnung pausieren/fortsetzen</string>
58               </property>
59              </widget>
60             </item>
61            </layout>
62           </widget>
63          </item>
64          <item row="5" column="0">
65           <widget class="Line" name="line1">
66            <property name="orientation">
67             <enum>Qt::Horizontal</enum>
68            </property>
69           </widget>
70          </item>
71          <item row="6" column="0">
72           <widget class="QLabel" name="lblCurrAcc">
73            <property name="text">
74             <string>Aktuelle Genauigkeit: n/a</string>
75            </property>
76            <property name="alignment">
77             <set>Qt::AlignCenter</set>
78            </property>
79           </widget>
```

```
80          </item>
81          <item row="7" column="0">
82           <widget class="QLabel" name="lblCurrIter">
83            <property name="text">
84             <string>Aktuelle Anzahl Iterationsschritte: n/a</string>
85            </property>
86            <property name="alignment">
87             <set>Qt::AlignCenter</set>
88            </property>
89           </widget>
90          </item>
91          <item row="8" column="0">
92           <widget class="Line" name="line2">
93            <property name="orientation">
94             <enum>Qt::Horizontal</enum>
95            </property>
96           </widget>
97          </item>
98          <item row="9" column="0">
99           <widget class="QPlainTextEdit" name="plTextEditLog">
100           <property name="verticalScrollBarPolicy">
101            <enum>Qt::ScrollBarAlwaysOn</enum>
102           </property>
103           <property name="lineWrapMode">
104            <enum>QPlainTextEdit::WidgetWidth</enum>
105           </property>
106           <property name="readOnly">
107            <bool>true</bool>
108           </property>
109          </widget>
110         </item>
111         <item row="0" column="0">
112          <widget class="QGroupBox" name="grpBoxGleichung">
113           <property name="title">
114            <string>Konvektions-Diffusionsgleichung</string>
115           </property>
116           <layout class="QFormLayout" name="formLayout">
117            <item row="0" column="0">
118             <widget class="QLabel" name="lblEpsilon">
119              <property name="text">
120               <string>e:</string>
121              </property>
122             </widget>
123            </item>
124            <item row="0" column="1">
125             <widget class="QLineEdit" name="leEpsilon">
126              <property name="toolTip">
127               <string>&amp;lt;!DOCTYPE HTML PUBLIC &amp;quot;-//W3C//DTD HTML 4.0//
                    EN&amp;quot; &amp;quot;http://www.w3.org/TR/REC-html40/strict.dtd&
                    amp;quot;&amp;gt;
128 &amp;lt;html&amp;gt;&amp;lt;head&amp;gt;&amp;lt;meta name=&amp;quot;qrichtext&amp;quot; content=&amp;quot
    ;1&amp;quot; /&amp;gt;&amp;lt;style type=&amp;quot;text/css&amp;quot;&amp;gt;
```

```
129 p, li { white-space: pre-wrap; }
130 &lt;/style&gt;&lt;/head&gt;&lt;body style=&quot; font-family:'MS Shell Dlg
      2'; font-size:7.8pt; font-weight:400; font-style:normal;&quot;&gt;
131 &lt;p style=&quot; margin-top:0px; margin-bottom:0px; margin-left:0px;
    margin-right:0px; -qt-block-indent:0; text-indent:0px;&quot;&gt;&lt;
    span style=&quot; font-size:8pt;&quot;&gt;e entspricht dem
    Diffusionskoeffizienten innerhalb der Konvektions-Diffusionsgleichung.&
    lt;/span&gt;&lt;/p&gt;&lt;/body&gt;&lt;/html&gt;</string>
132             </property>
133             <property name="text">
134              <string>0.001</string>
135             </property>
136            </widget>
137           </item>
138           <item row="1" column="0">
139            <widget class="QLabel" name="lblBeta">
140             <property name="text">
141              <string>ß:</string>
142             </property>
143            </widget>
144           </item>
145           <item row="1" column="1">
146            <layout class="QHBoxLayout" name="layout3">
147             <item>
148              <widget class="QLineEdit" name="leBeta1">
149               <property name="toolTip">
150                <string>Die Stärke des Konvektionsterms in x-Richtung.</
                     string>
151               </property>
152               <property name="text">
153                <string>1.0</string>
154               </property>
155              </widget>
156             </item>
157             <item>
158              <widget class="QLineEdit" name="leBeta2">
159               <property name="toolTip">
160                <string>Die Stärke des Konvektionsterms in y-Richtung.</
                     string>
161               </property>
162               <property name="text">
163                <string>1.0</string>
164               </property>
165              </widget>
166             </item>
167            </layout>
168           </item>
169          </layout>
170         </widget>
171        </item>
172        <item row="1" column="0">
173         <widget class="QGroupBox" name="grpBoxDiskretisierung">
```

```
174          <property name="title">
175           <string>Diskretisierung</string>
176          </property>
177          <layout class="QVBoxLayout" name="verticalLayout">
178           <item>
179            <layout class="QHBoxLayout" name="layout2">
180             <item>
181              <widget class="QLabel" name="lblN">
182               <property name="text">
183                <string>n:</string>
184               </property>
185              </widget>
186             </item>
187             <item>
188              <widget class="QLineEdit" name="leN">
189               <property name="toolTip">
190                <string>Die Auflösung des Diskretisierungsgitters.</string>
191               </property>
192               <property name="text">
193                <string>100</string>
194               </property>
195              </widget>
196             </item>
197            </layout>
198           </item>
199           <item>
200            <layout class="QHBoxLayout" name="layout1">
201             <item>
202              <widget class="QLabel" name="lblH">
203               <property name="text">
204                <string>h:</string>
205               </property>
206              </widget>
207             </item>
208             <item>
209              <widget class="QLineEdit" name="leH">
210               <property name="toolTip">
211                <string>Die Gitterweite des Diskretisierungsgitters. Sie
                     resultiert direkt aus der Auflösung n.</string>
212               </property>
213               <property name="text">
214                <string>0.01</string>
215               </property>
216               <property name="readOnly">
217                <bool>true</bool>
218               </property>
219              </widget>
220             </item>
221            </layout>
222           </item>
223           <item>
224            <widget class="QCheckBox" name="chkBoxUseUpwinding">
```

```
225             <property name="toolTip">
226              <string>Bestimmt ob Upwind-Diskretisierungen statt der
                     symmetrischen Differenzenquotienten verwendet werden
                     sollen. Die numerische Stabilität wird mit einem höheren
                     Diskretisierungsfehler bezahlt.</string>
227             </property>
228             <property name="text">
229              <string>Upwind-Diskretisierung verwenden</string>
230             </property>
231             <property name="checked">
232              <bool>true</bool>
233             </property>
234            </widget>
235           </item>
236          </layout>
237         </widget>
238        </item>
239        <item row="2" column="0">
240         <widget class="QGroupBox" name="grpBoxIteration">
241          <property name="title">
242           <string>Iteration</string>
243          </property>
244          <layout class="QFormLayout" name="formLayout_5">
245           <item row="0" column="0">
246            <widget class="QLabel" name="lblE">
247             <property name="text">
248              <string>e:</string>
249             </property>
250            </widget>
251           </item>
252           <item row="1" column="0">
253            <widget class="QLabel" name="lblOmega">
254             <property name="text">
255              <string>?:</string>
256             </property>
257            </widget>
258           </item>
259           <item row="2" column="0">
260            <widget class="QLabel" name="lblGamma">
261             <property name="text">
262              <string>?:</string>
263             </property>
264            </widget>
265           </item>
266           <item row="0" column="1">
267            <widget class="QLineEdit" name="leE">
268             <property name="toolTip">
269              <string>Die Genauigkeit, bis zu der gerechnet werden soll.
                     Das Verfahren wird abgebrochen, sobald das Residuum die
                     gewünschte Genauigkeit erreicht hat.</string>
270             </property>
271             <property name="text">
```

```
272                 <string>1e-4</string>
273                </property>
274               </widget>
275              </item>
276              <item row="1" column="1">
277               <widget class="QLineEdit" name="leOmega">
278                <property name="toolTip">
279                 <string>&lt;!DOCTYPE HTML PUBLIC &quot;-//W3C//DTD HTML 4.0//
                        EN&quot; &quot;http://www.w3.org/TR/REC-html40/strict.dtd&
                        quot;&gt;
280 &lt;html&gt;&lt;head&gt;&lt;meta name=&quot;qrichtext&quot; content=&quot
    ;1&quot; /&gt;&lt;style type=&quot;text/css&quot;&gt;
281 p, li { white-space: pre-wrap; }
282 &lt;/style&gt;&lt;/head&gt;&lt;body style=&quot; font-family:'MS Shell Dlg
     2'; font-size:7.8pt; font-weight:400; font-style:normal;&quot;&gt;
283 &lt;p style=&quot; margin-top:0px; margin-bottom:0px; margin-left:0px;
    margin-right:0px; -qt-block-indent:0; text-indent:0px;&quot;&gt;&lt;
    span style=&quot; font-size:8pt;&quot;&gt;Dies ist der
    Dämpfungsparameter für das gedämpfte Richardson-Verfahren. Es kann mit
    Werten 0 &amp;lt; ? &amp;lt; 2 über bzw. unterrelaxiert werden, wobei
    ein Wert von ?=1.0 dem stationären Richardson-Verfahren entspricht. &lt
    ;/span&gt;&lt;/p&gt;&lt;/body&gt;&lt;/html&gt;</string>
284                </property>
285                <property name="text">
286                 <string>1.0</string>
287                </property>
288               </widget>
289              </item>
290              <item row="2" column="1">
291               <widget class="QLineEdit" name="leGamma">
292                <property name="toolTip">
293                 <string>Dieser Parameter sorgt für eine Über- bzw.
                        Unterrelaxation der Hauptdiagonaleinträge. </string>
294                </property>
295                <property name="text">
296                 <string>0.0</string>
297                </property>
298               </widget>
299              </item>
300             </layout>
301            </widget>
302           </item>
303           <item row="3" column="0">
304            <widget class="QGroupBox" name="grpBoxOutput">
305             <property name="title">
306              <string>Ausgabe</string>
307             </property>
308             <layout class="QVBoxLayout" name="verticalLayout_2">
309              <item>
310               <widget class="QCheckBox" name="chkBoxPlot3D">
311                <property name="toolTip">
312                 <string>Bestimmt ob die dreidimensionale Visualisierung
```

```
            geplottet und gespeichert werden soll. Es wird
            voausgesetzt, dass gnuplot installiert und über die
            Kommandozeile erreichbar ist.</string>
313       </property>
314       <property name="text">
315        <string>Dreidimensionale Visualisierung plotten</string>
316       </property>
317       <property name="checked">
318        <bool>false</bool>
319       </property>
320      </widget>
321     </item>
322     <item>
323      <layout class="QHBoxLayout" name="layout4" stretch="3,0">
324       <item>
325        <widget class="QLineEdit" name="le3DPlotFileName">
326         <property name="enabled">
327          <bool>false</bool>
328         </property>
329         <property name="sizePolicy">
330          <sizepolicy hsizetype="Expanding" vsizetype="Ignored">
331           <horstretch>0</horstretch>
332           <verstretch>0</verstretch>
333          </sizepolicy>
334         </property>
335         <property name="toolTip">
336          <string>Der Dateipfad, unter dem die dreidimensionale
               Visualisierung gespeichert werden soll.</string>
337         </property>
338         <property name="text">
339          <string>C:/Plot.pdf</string>
340         </property>
341        </widget>
342       </item>
343       <item>
344        <widget class="QPushButton" name="btnFindFile">
345         <property name="enabled">
346          <bool>false</bool>
347         </property>
348         <property name="sizePolicy">
349          <sizepolicy hsizetype="Preferred" vsizetype="Preferred">
350           <horstretch>0</horstretch>
351           <verstretch>0</verstretch>
352          </sizepolicy>
353         </property>
354         <property name="minimumSize">
355          <size>
356           <width>0</width>
357           <height>0</height>
358          </size>
359         </property>
360         <property name="baseSize">
```

```
361                 <size>
362                  <width>0</width>
363                  <height>0</height>
364                 </size>
365                </property>
366                <property name="toolTip">
367                 <string/>
368                </property>
369                <property name="text">
370                 <string>Durchsuchen</string>
371                </property>
372               </widget>
373              </item>
374             </layout>
375            </item>
376           </layout>
377          </widget>
378         </item>
379        </layout>
380       </widget>
381      </item>
382      <item row="0" column="1">
383       <widget class="QGraphicsView" name="graphicsView">
384        <property name="palette">
385         <palette>
386          <active>
387           <colorrole role="Base">
388            <brush brushstyle="SolidPattern">
389             <color alpha="0">
390              <red>255</red>
391              <green>255</green>
392              <blue>255</blue>
393             </color>
394            </brush>
395           </colorrole>
396          </active>
397          <inactive>
398           <colorrole role="Base">
399            <brush brushstyle="SolidPattern">
400             <color alpha="0">
401              <red>255</red>
402              <green>255</green>
403              <blue>255</blue>
404             </color>
405            </brush>
406           </colorrole>
407          </inactive>
408          <disabled>
409           <colorrole role="Base">
410            <brush brushstyle="SolidPattern">
411             <color alpha="255">
412              <red>240</red>
```

```
413                <green>240</green>
414                <blue>240</blue>
415               </color>
416              </brush>
417             </colorrole>
418            </disabled>
419           </palette>
420          </property>
421          <property name="autoFillBackground">
422           <bool>false</bool>
423          </property>
424          <property name="styleSheet">
425           <string notr="true"/>
426          </property>
427          <property name="frameShape">
428           <enum>QFrame::Box</enum>
429          </property>
430          <property name="frameShadow">
431           <enum>QFrame::Sunken</enum>
432          </property>
433          <property name="backgroundBrush">
434           <brush brushstyle="NoBrush">
435            <color alpha="255">
436             <red>0</red>
437             <green>0</green>
438             <blue>0</blue>
439            </color>
440           </brush>
441          </property>
442          <property name="foregroundBrush">
443           <brush brushstyle="NoBrush">
444            <color alpha="255">
445             <red>0</red>
446             <green>0</green>
447             <blue>0</blue>
448            </color>
449           </brush>
450          </property>
451         </widget>
452        </item>
453       </layout>
454      </widget>
455      <widget class="QMenuBar" name="menuBar">
456       <property name="geometry">
457        <rect>
458         <x>0</x>
459         <y>0</y>
460         <width>761</width>
461         <height>26</height>
462        </rect>
463       </property>
464       <widget class="QMenu" name="menuProgramm">
```

```
465      <property name="title">
466       <string>Programm</string>
467      </property>
468      <addaction name="actionBeenden"/>
469     </widget>
470     <widget class="QMenu" name="menuHilfe">
471      <property name="title">
472       <string>Hilfe</string>
473      </property>
474      <addaction name="actionUeber"/>
475     </widget>
476     <addaction name="menuProgramm"/>
477     <addaction name="menuHilfe"/>
478    </widget>
479    <action name="actionBeenden">
480     <property name="text">
481      <string>Beenden</string>
482     </property>
483    </action>
484    <action name="actionUeber">
485     <property name="text">
486      <string>Über...</string>
487     </property>
488    </action>
489   </widget>
490   <layoutdefault spacing="6" margin="11"/>
491   <resources>
492    <include location="Ressourcen.qrc"/>
493   </resources>
494   <connections/>
495  </ui>
```

Listing A.4: mainwindow.ui

```
1  #ifndef MATRIX_H
2  #define MATRIX_H
3
4  #include <QString>
5
6  /**
7   * @brief Hilfskonstrukt fuer die Klasse Matrix.
8   *
9   * Dies ist ein Hilfskonstrukt fuer die Matrix selbst. Sie stellt einen
10  * einzelnen Eintrag in selbiger dar.
11  */
12 struct Entry {
13   /**
14    * Die Zeile des Eintrags.
15    */
16   int row;
17
18   /**
19    * Die Spalte des Eintrags.
```

```
20      */
21     int col;
22
23     /**
24      * Der Wert in dem Eintrag.
25      */
26     float val;
27
28     /**
29      * Erzeugt eine neue Instanz dieses structs mit den gegebenen
30      * Parametern.
31      *
32      * @param row
33      *              Die Zeile, in welcher sich dieser Eintrag befindet.
34      * @param col
35      *              Die Spalte, in welcher sich dieser Eintrag befindet.
36      * @param val
37      *              Der Wert in dem Eintrag.
38      */
39     Entry(int row, int col, float val);
40 };
41
42 /**
43  * @brief Eine spezifische Matrix-Klasse.
44  *
45  * Diese Klasse repraesentiert eine sehr spezifische duennbesetzte Matrix.
46  * Die Matrix selbst haette eigentlich (n+1)^2 x (n+1)^2 Eintraege, aber
47  * um Nulleintraege einzusparen, werden lediglich die  von Null
48  * verschiedenen Eintraege gespeichert. Dazu macht sich dieser Datentyp zu
49  * Nutze, dass sowohl in jeder Zeile, als auch in jeder Spalte lediglich
50  * fuenf Eintraege existieren. Um gewisse Eigenschaften fuer eine hohe
51  * Performance sicherzustellen, werden die meisten Methoden dieser Klasse
52  * gekapselt und koennen lediglich von dem Arbeitsthread in dem gleichen
53  * Package genutzt werden.
54  *
55  * @author Nils Christian Ehmke
56  */
57 class Matrix {
58 public:
59     /**
60      * Der Destruktor fuer diese Klasse.
61      */
62     ~Matrix();
63
64     /**
65      * Diese Methode liefert bei gegebenen Parametern die korrekte
66      * Koeffizientenmatrix fuer die zugrunde liegende Gleichung.
67      *
68      * @param n
69      *              Definiert die Aufloesung des Diskretisierungsgitters.
70      * @param beta1
71      *              Erster Parameter fuer den Konvektionsterm.
```

```
72      * @param beta2
73      *             Zweiter Parameter fuer den Konvektionsterm.
74      * @param epsilon
75      *             Der Diffusionskoeffizient.
76      * @param useUpwinding
77      *             Bestimmt ob die Upwind-Diskretisierung verwendet werden
78      *             soll oder nicht.
79      * @return Eine geeignete Koeffizientenmatrix fuer die gegebenen
80      *         Parameter.
81      */
82     static Matrix *getMatrix(int n, double beta1, double beta2,
83                              double epsilon, bool useUpwinding);
84
85     /**
86      * Diese Methode berechnet die Maximumsnorm des gegebenen Vektors.
87      *
88      * @param vec
89      *             Der Vektor.
90      * @param size
91      *             Die Groesse des Vektors.
92      * @return Die Maximumsnorm des Vektors.
93      */
94     static double infinityNorm(double vec[], int size);
95
96     /**
97      * Diese Methode subtrahiert zwei Vektoren. Das Ergebnis wird direkt
98      * in den ersten Vektor geschrieben. Es handelt sich also um ein
99      * destruktives Update.
100     *
101     * @param a
102     *             Der erste Vektor. Er dient gleichzeitig als
103     *             Ergebnisvektor.
104     * @param b
105     *             Der zweite Vektor.
106     * @param size
107     *             Die Groesse der Vektoren.
108     */
109    static void sub(double a[], double b[], int size);
110
111    /**
112     * Diese Methode multipliziert den gegebenen Vektor mit dem gegebenen
113     * Skalar.
114     *
115     * @param omega
116     *                Der zu multiplizierende Vektor.
117     * @param a
118     *                Der Vektor. Das Ergebnis wird direkt in diesen
119     *                hineingeschrieben.
120     * @param size
121     *                Die Groesse des Vektors.
122     */
123    static void multiply(double omega, double a[], int size);
```

```
124
125     /**
126      * Diese Methode liefert den Wert an einer gegeben Position. Falls die
127      * Position nicht innerhalb der Besetzungsstruktur der Matrix ist, so
128      * wird 0.0 zurueckgelifert.<br>
129      * Beide Parameter fuer Zeile und Spalte sollten indiziert sein von 0
130      * bis (n+1)^2 - 1 einschliesslich.
131      *
132      * @param row
133      *             Der Zeilenindex.
134      * @param column
135      *             Der Spaltenindex.
136      * @return Der aktuelle Wert bei A[row][column] oder 0.0, falls die
137      *         Position nicht gefuellt ist.
138      */
139     float getValue(int row, int column);
140
141     /**
142      * Diese Methode berechnet die ILU-Zerlegung der aktuellen Matrix und
143      * liefert diese innerhalb einer neuen Instanz dieser Klasse zurueck,
144      * wobei die Dreiecksmatrizen L und U zusammen in einem Objekt
145      * gespeichert werden.
146      *
147      * @return Eine Matrix, welche die ILU-Zerlegung der aktuellen Instanz
148      *         beinhaltet.
149      */
150     Matrix *getILU();
151
152     /**
153      * Diese Methode multipliziert einen gegeben Vektor mit der aktuellen
154      * Matrix. Es wird nicht ueberprueft, ob die Dimensionen passend sind
155      * oder nicht.
156      *
157      * @param vector
158      *             Der Vektor, der mit dieser Matrix multipliziert
159      *             werden soll.
160      * @return Das resultierende Matrix-Vektor-Produkt.
161      */
162     void multiply(double vector[]);
163
164     /**
165      * Diese Methode interpretiert die aktuelle Instanz als ILU-Zerlegung
166      * und multipliziert das Inverse des Produkts der beiden Matrizen mit
167      * dem gegebenen Vektor. Die Multiplikation wird allerdings in
168      * Wirklichkeit mithilfe von Vorwaerts-/Rueckwaertseinsetzen effizient
169      * geloest.
170      *
171      * @param y
172      *             Der zu multiplizierende Vektor.
173      * @return (LU)^-1 * y, sofern die aktuelle Matrix die Zerlegung mit
174      *         den Matrizen L und U repraesentiert.
175      */
```

```
176     void multiplyWithInverseAsILU(double y[]);
177
178     /**
179      * Diese Methode liefert eine Stringrepraesentation der Matrix.
180      *
181      * @return Ein String, welcher die Matrix darstellt.
182      */
183     QString toString();
184
185     /**
186      * Diese Methode sorgt fuer eine ueber-/Unterrelaxation der
187      * Hauptdiagonaleintraege der aktuellen Matrix. Ein Aufruf mit 0.0
188      * als Faktor wird ignoriert.
189      *
190      * @param gamma
191      *            Der Relaxationsparameter.
192      */
193     void addRelaxation(double gamma);
194
195 private:
196     /**
197      * Die Aufloesung des Diskretisierungsgitters.
198      */
199     int n;
200
201     /**
202      * Die "reale" Groesse der Matrix.
203      */
204     int dim;
205
206     /**
207      * Das dynamische Array fuer den zeilenweisen Zugriff.
208      */
209     Entry ***rowAcc;
210
211     /**
212      * Das dynamische Array fuer den spaltenweisen Zugriff.
213      */
214     Entry ***colAcc;
215
216     /**
217      * Erzeugt eine neue Instanz dieser Klasse mit dem gegebenen
218      * Parameter.
219      *
220      * @param n
221      *            Die "Groesse" des Problems. Sie wird genutzt, um die
222      *            korrekte Groesse der Matrix zu berechnen.
223      */
224     Matrix(int n);
225
226     /**
227      * Diese Methode setzt einen Wert an der gegebenen Position. Diese
```

```
228     * Methode sollte fuer die Matrix von links nach rechts und von oben
229     * nach unten aufgerufen werden, damit die Eintraege an der korrekten
230     * Position in dem Array stehen.<br>
231     * Beide Parameter fuer zeile und Spalte sollten indiziert sein von 0
232     * bis (n+1)^2 - 1 einschliesslich. Fehlerhafte Indizierung wird
233     * ignoriert.
234     *
235     * @param row
236     *            Die Position der Zeile, die zu setzen ist.
237     * @param col
238     *            Die Position der Spalte, die zu setzen ist.
239     * @param val
240     *            Der Wert fuer die gegebene Position.
241     */
242    void setValue(int row, int col, float val);
243
244    /**
245     * Diese Methode addiert einen Wert an der gegebenen Position. Ist der
246     * Eintrag noch nicht vorhanden, so wird er angelegt. In diesem Fall
247     * sollte diese Methode fuer die Matrix von links nach rechts und von
248     * oben nach unten aufgerufen werden, damit die Eintraege an der
249     * korrekten Position in dem Array stehen.<br>
250     * Beide Parameter fuer zeile und Spalte sollten indiziert sein von 0
251     * bis (n+1)^2 - 1 einschliesslich. Fehlerhafte Indizierung wird
252     * ignoriert.
253     *
254     * @param row
255     *            Die Position der Zeile, die zu setzen ist.
256     * @param col
257     *            Die Position der Spalte, die zu setzen ist.
258     * @param val
259     *            Der zu addierende Wert fuer die gegebene Position.
260     */
261    void addValue(int row, int col, float val);
262
263    /**
264     * Diese Methode sucht nach dem gegebenen Eintrag und loescht ihn aus
265     * der Matrix.
266     *
267     * @param row
268     *            Die Position der Zeile, die zu loeschen ist.
269     * @param col
270     *            Die Position der Spalte, die zu loeschen ist.
271     */
272    void remove(int row, int col);
273 };
274
275
276
277 #endif // MATRIX_H
```

Listing A.5: matrix.h

```
1  #include "matrix.h"
2  #include <math.h>
3
4  Entry::Entry(int row, int col, float val) {
5    this->row = row;
6    this->col = col;
7    this->val = val;
8  }
9
10 Matrix::Matrix(int n) {
11   /* Speichere die Abmessungen der Matrix */
12   this->n = n;
13   this->dim = (n - 1) * (n - 1);
14
15   /* Initialisiere die Zugriffsfelder */
16   this->rowAcc = new Entry**[dim];
17   this->colAcc = new Entry**[dim];
18
19   for (int i = 0; i < dim; i++) {
20     this->rowAcc[i] = new Entry*[5];
21     this->colAcc[i] = new Entry*[5];
22
23     for (int j = 0; j < 5; j++) {
24       this->rowAcc[i][j] = 0;
25       this->colAcc[i][j] = 0;
26     }
27   }
28 }
29
30 Matrix::~Matrix() {
31   /* Gebe nun alles wieder frei. */
32   for (int i = 0; i < dim; i++) {
33     for (int j = 0; j < 5; j++) {
34       /* Hier nur ein Element löschen, da Zeile und Spalte auf das
35          identische Objekt verweisen. */
36       if (this->rowAcc[i][j] != 0) {
37         delete this->rowAcc[i][j];
38       }
39     }
40     delete[] this->rowAcc[i];
41     delete[] this->colAcc[i];
42   }
43
44   delete[] this->rowAcc;
45   delete[] this->colAcc;
46 }
47
48 void Matrix::addRelaxation(double gamma) {
49   /* If the factor is 0.0, we can ignore this method call. */
50   if (gamma == 0.0){
51     return;
52   }
```

```
53    /* Dieser Bereich dient der Berechnung der Ueber-/Unterrelaxation */
54    for (int i = 0; i < dim; i++) {
55      float sum = 0.0;
56      for (int j = 0; j < 5; j++) {
57        Entry *e = rowAcc[i][j];
58        if (e != 0 && e->col != i) {
59          sum += fabs(e->val);
60        }
61      }
62      /* Addiere den zusätzlichen Wert auf den Hauptdiagonaleintrag. */
63      for (int j = 0; j < 5; j++) {
64        Entry *e = rowAcc[i][j];
65        if (e != 0 && e->col == i) {
66          e->val = e->val + gamma * sum;
67        }
68      }
69    }
70  }
71
72  Matrix *Matrix::getMatrix(int n, double beta1, double beta2,
73                            double epsilon, bool useUpwinding) {
74    /* Bereite die Matrix und die Gitterhoehe vor. */
75    Matrix *result = new Matrix(n);
76    float h = 1.0 / n;
77
78    /* Berechne die Werte fuer die Eintraege in der Matrix jetzt, damit sie
79       nicht staendig neuberechnet werden. */
80    float entry1 = 4.0 * epsilon / (h * h);
81    float entry2 = -1.0 * epsilon / (h * h);
82    float entry3 = beta1 / (2.0 * h);
83    float entry4 = beta2 / (2.0 * h);
84    int index;
85
86    /* Durchlaufe nun jedes Feld des Diskretisierungsgitter. Es
87       * verbleibt im Hinterkopf zu behalten, dass das eigentlich
88       * zweidimensionale Problem je auf eine eindimensionale "Matrix"
89       * umberechnet werden muss. */
90    for (int i = 0; i < n - 1; i++) {
91      for (int j = 0; j < n - 1; j++) {
92        index = i * (n - 1) + j;
93
94        if (useUpwinding) {
95          result->setValue(index, i * (n - 1) + j,
96                           entry1);
97          result->setValue(index, i * (n - 1) + (j + 1),
98                           entry2);
99          result->setValue(index, i * (n - 1) + (j - 1),
100                           entry2);
101         result->setValue(index, (i + 1) * (n - 1) + j,
102                          entry2);
103         result->setValue(index, (i - 1) * (n - 1) + j,
104                          entry2);
```

```
105
106         /* Hier die Fallunterscheidung für das Upwinding. */
107         if (beta1 < 0) {
108           result->addValue(index, (i + 1) * (n - 1) + j,
109                            beta1 / (1.0 * h));
110           result->addValue(index, i * (n - 1) + j,
111                            - beta1 / (1.0 * h));
112         } else {
113           result->addValue(index, (i - 1) * (n - 1) + j,
114                            - beta1 / (1.0 * h));
115           result->addValue(index, i * (n - 1) + j,
116                            beta1 / (1.0 * h));
117         }
118
119         if (beta2 < 0) {
120           result->addValue(index, i * (n - 1) + (j + 1),
121                            beta2 / (1.0 * h));
122           result->addValue(index, i * (n - 1) + j,
123                            - beta2 / (1.0 * h));
124         } else {
125           result->addValue(index, i * (n - 1) + (j - 1),
126                            - beta2 / (1.0 * h));
127           result->addValue(index, i * (n - 1) + j,
128                            beta2 / (1.0 * h));
129         }
130
131       } else {
132         result->setValue(index, i * (n - 1) + j,
133                          entry1);
134         result->setValue(index, i * (n - 1) + (j + 1),
135                          entry2 + entry4);
136         result->setValue(index, i * (n - 1) + (j - 1),
137                          entry2 - entry4);
138         result->setValue(index, (i + 1) * (n - 1) + j,
139                          entry2 + entry3);
140         result->setValue(index, (i - 1) * (n - 1) + j,
141                          entry2 - entry3);
142       }
143
144       /* Die ungültigen Einträge am Rand wieder löschen */
145       for (int k = (n-1); k < (n-1) * (n-1); k += (n-1)) {
146         result->remove(k, k - 1);
147         result->remove(k - 1, k);
148       }
149     }
150   }
151
152   return result;
153 }
154
155 float Matrix::getValue(int row, int column) {
156   /* Durchlaufe alle Eintraege in der Zeile und suche nach dem, der den
```

```
157       richtigen Spaltenindex hat. */
158     for (int k = 0; k < 5; k++) {
159       if (rowAcc[row][k] != 0 && rowAcc[row][k]->col == column) {
160         /* Gefunden. Liefere den Wert zurueck. */
161         return rowAcc[row][k]->val;
162       }
163     }
164     /* Nichts gefunden. Liefere 0.0 zurueck. */
165     return 0.0;
166 }
167
168 void Matrix::remove(int row, int col) {
169     /* Durchlaufe alle Eintraege in der Zeile und suche nach dem, der den
170        richtigen Spaltenindex hat. */
171     for (int a = 0; a < 5; a++) {
172       if (rowAcc[row][a] != 0) {
173         if (rowAcc[row][a]->col == col) {
174           /* Lösche den Eintrag und verschiebe die nächsten nach links. */
175           delete rowAcc[row][a];
176           rowAcc[row][a] = 0;
177           for (int b = a + 1; b < 5; b++) {
178             rowAcc[row][b - 1] = rowAcc[row][b];
179           }
180           rowAcc[row][4] = 0;
181           /* Springe aus der Methode. */
182           return;
183         }
184       }
185     }
186 }
187
188 void Matrix::setValue(int row, int col, float val) {
189     /* Ignoriere ungueltige Indizierung. */
190     if (row < 0 || row >= dim || col < 0 || col >= dim) {
191       return;
192     }
193
194     /* Es wird davon ausgegangen, dass jedes Feld nur einmal initialisiert
195        wird! */
196     for (int a = 0; a < 5; a++) {
197       if (rowAcc[row][a] == 0) {
198         rowAcc[row][a] = new Entry(row, col, val);
199         /* Weise die Referenz nun auch colAcc zu. */
200         for (int b = 0; b < 5; b++) {
201           if (colAcc[col][b] == 0) {
202             colAcc[col][b] = rowAcc[row][a];
203             return;
204           }
205         }
206       }
207     }
208 }
```

```
209
210 void Matrix::addValue(int row, int col, float val) {
211   /* Ignoriere ungueltige Indizierung. */
212   if (row < 0 || row >= dim || col < 0 || col >= dim) {
213     return;
214   }
215
216   /* Versuche das Feld zunächst zu finden. */
217   for (int a = 0; a < 5; a++) {
218     if (rowAcc[row][a] != 0 && rowAcc[row][a]->col == col) {
219       /* Eintrag gefunden. Addiere den Wert und verlasse die Methode. */
220       rowAcc[row][a]->val += val;
221       return;
222     }
223   }
224
225   /* Wenn wir hier sind, so wurde der Eintrag noch nicht angelegt. */
226   setValue(row, col, val);
227 }
228
229 double Matrix::infinityNorm(double vec[], int size) {
230   double m = fabs(vec[0]);
231
232   /* Durchlaufe alle Komponenten und finde den betragsmaessig groessten
233      Eintrag. */
234   for (int i = 1; i < size; i++) {
235     m = fmax(m, fabs(vec[i]));
236   }
237
238   return m;
239 }
240
241 void Matrix::sub(double a[], double b[], int size) {
242   /* Subtrahiere komponentenweise. */
243   for (int i = 0; i <  size; i++) {
244     a[i] -= b[i];
245   }
246 }
247
248 void Matrix::multiply(double omega, double a[], int size) {
249   for (int i = 0; i <  size; i++) {
250     a[i] *= omega;
251   }
252 }
253
254 void Matrix::multiply(double vector[]) {
255   double *temp = new double[dim];
256
257   /* Durchlaufe jede Komponente des gegebenen Vektors. */
258   for (int i = 0; i < dim; i++) {
259     double sum = 0.0;
260
```

```
261       /* Durchlaufe nur alle von Null verschiedenen Eintraege in der
262          aktuellen Zeile. */
263       for (int k = 0; k < 5; k++) {
264         Entry *currEntry = rowAcc[i][k];
265         if (currEntry != 0) {
266           sum += currEntry->val * vector[currEntry->col];
267         }
268       }
269
270       /* Eintrag berechnet. */
271       temp[i] = sum;
272     }
273
274     /* Kopiere das Ergebnis. */
275     for (int i = 0; i < dim; i++) {
276       vector[i] = temp[i];
277     }
278
279     delete[] temp;
280   }
281
282   Matrix *Matrix::getILU() {
283     Matrix *matrix = new Matrix(n);
284
285     /* Kopiere die Matrix zunaechst. */
286     int size = (n - 1) * (n - 1);
287     for (int i = 0; i < size; i++) {
288       for (int k = 0; k < 5; k++) {
289         if (this->rowAcc[i][k] != 0) {
290           matrix->setValue(this->rowAcc[i][k]->row,
291                            this->rowAcc[i][k]->col,
292                            this->rowAcc[i][k]->val);
293           matrix->rowAcc[i][k]->row = this->rowAcc[i][k]->row;
294         }
295       }
296     }
297
298     /* Verwende nun den Algorithmus fuer die ILU-Zerlegung, wobei aber nur
299        die von Null verschiedenen Eintraege durchlaufen werden, um Zeit zu
300        sparen. */
301     int j;
302
303     for (int k = 1; k <= size - 1; k++) {
304       for (int d2 = 0; d2 < 5; d2++) {
305         Entry *sEntry = matrix->colAcc[k - 1][d2];
306         if (sEntry != 0) {
307           int i = sEntry->row + 1;
308           if (i > size || i < k + 1) {
309             continue;
310           }
311           if (sEntry != 0 && sEntry->val != 0.0) {
312             sEntry->val /= matrix->getValue(k - 1, k - 1);
```

```
313
314             for (int d = 0; d < 5; d++) {
315               Entry *currEntry = matrix->rowAcc[i - 1][d];
316               if (currEntry != 0
317                   && currEntry->col < size
318                   && currEntry->col >= k) {
319                 j = currEntry->col + 1;
320                 currEntry->val -= sEntry->val
321                                   * matrix->getValue(k - 1,
322                                                      j - 1);
323               }
324             }
325           }
326         }
327       }
328     }
329
330     return matrix;
331 }
332
333 void Matrix::multiplyWithInverseAsILU(double y[]) {
334     /* Vorwaertseinsetzen. */
335     double *r = new double[dim];
336
337     for (int i = 1; i <= dim; i++) {
338       double sum = 0.0;
339
340       for (int d = 0; d < 5; d++) {
341         Entry *currEntry = rowAcc[i - 1][d];
342         if (currEntry != 0 && currEntry->col >= 0
343             && currEntry->col < i - 1) {
344           sum += currEntry->val * r[currEntry->col];
345         }
346       }
347
348       /* Da die Eintraege auf der Hauptdiagonale von L alle eins sind,
349          muessen wir 1/a_11 nicht noch multiplizieren. */
350       r[i - 1] = y[i - 1] - sum;
351     }
352
353     /* Rueckwaertseinsetzen */
354     double *z = new double[dim];
355     for (int i = dim; i >= 1; i--) {
356       double sum = 0.0;
357
358       for (int d = 0; d < 5; d++) {
359         Entry *currEntry = rowAcc[i - 1][d];
360         if (currEntry != 0 && currEntry->col < dim
361             && currEntry->col >= i) {
362           sum += currEntry->val * z[currEntry->col];
363         }
364       }
```

```
365
366       z[i - 1] = 1.0 / getValue(i - 1, i - 1)
367                   * (r[i - 1] - sum);
368     }
369
370     /* Ergebnis kopieren. */
371
372     for (int i = 0; i < dim; i++) {
373       y[i] = z[i];
374     }
375
376     delete[] r;
377     delete[] z;
378 }
379
380 QString Matrix::toString() {
381     QString str = "";
382
383     for (int i = 0; i < dim; i++) {
384       for (int j = 0; j < dim; j++) {
385         str += "[" + QString::number(getValue(i, j)) + "]";
386       }
387       str += "\n";
388     }
389
390     return str;
391 }
```

Listing A.6: matrix.cpp

```
1  #ifndef WORKTHREAD_H
2  #define WORKTHREAD_H
3
4  #include <QThread>
5  #include <QPixmap>
6  #include <matrix.h>
7  #include <QGraphicsScene>
8  #include <QMutex>
9  #include <QWaitCondition>
10
11 /**
12  * @brief Ein Arbeitsthread, der die numerische Berechnung uebernimmt.
13  *
14  * Diese Klasse stellt den Arbeitsthread dar, welcher die eigentliche
15  * Berechnung uebernehmen wird. Diese Berechnung befindet sich in einem
16  * speziellen Thread, um die GUI nicht zu blockieren. Zur Berechnung
17  * werden bestimmte Methoden der Klasse Matrix verwendet.<br>
18  * Die Klasse bekommt die Parameter fuer die Berechnung der
19  * Konvektions-Diffusions-Gleichung im Konstruktor mitgeliefert - die
20  * Berechnung wird dann wie ueblich mit start() begonnen. Sobald die
21  * Berechnung abgeschlossen ist, wird sie visualisiert und auf der GUI
22  * gezeichnet.<br>
23  * Weiterhin stellt der Thread eine sichere Methode bereit, um die Arbeit
```

```
24  * vorzeitig abzubrechen.
25  *
26  * @author Nils Christian Ehmke
27  */
28 class Workthread : public QThread {
29
30   Q_OBJECT
31
32 private:
33   /**
34    * Der Epsilon-Faktor innerhalb der Gleichung.
35    */
36   double epsilon;
37
38   /**
39    * Die erste Komponente des Beta-Vektors innerhalb der Gleichung.
40    */
41   double beta1;
42
43   /**
44    * Die zweite Komponente des Beta-Vektors innerhalb der Gleichung.
45    */
46   double beta2;
47
48   /**
49    * Der Schrankenwert fuer die Defektkorektur.
50    */
51   double e;
52
53   /**
54    * Die Aufloesung der Diskretisierungsgitters.
55    */
56   int n;
57
58   /**
59    * Der Daempfungsparameter fuer die Richardson-Iteration.
60    */
61   double omega;
62
63   /**
64    * Der Parameter fuer die ueber-/Unterrelaxation.
65    */
66   double gamma;
67
68   /**
69    * Die Leinwand, auf der gezeichnet wird.
70    */
71   QPixmap *pixmap;
72
73   /**
74    * Flag, ob der Arbeitsthread gestoppt wurde.
75    */
```

```
76    bool stopped;
77
78    /**
79     * Semaphore, um Zugriffe zu synchronisieren.
80     */
81    QMutex mutex;
82
83    /**
84     * Objekt um den Thread zu pausieren und fortfahren zu lassen.
85     */
86    QWaitCondition wcondition;
87
88    /**
89     * Flag, ob der Arbeitsthread pausiert wurde.
90     */
91    bool paused;
92
93    /**
94     * Dateiname fuer die dreidimensionale Visualisierung.
95     */
96    QString fileName3DPlot;
97
98    /**
99     * Bestimmt ob die Upwind-Diskretisierung verwendet werden soll oder
          nicht.
100    */
101   bool useUpwinding;
102
103
104   /**
105    * Diese Methode verwendet die Richardson Iteration, um die Loesung
106    * des LGS zu berechnen, wobei die gegebene Matrix als
107    * Koeffizientenmatrix genutzt wird. Weiterhin verwendet diese Methode
108    * die ILU-Zerlegung als Vorkonditionierer.
109    *
110    * @param m
111    *             Die zu verwendende Matrix.
112    * @param x0
113    *             Die Startloesung.
114    * @param e
115    *             Der Schrankenwert fuer die Defektkorektur.
116    * @param b
117    *             Die rechte Seite des LGS.
118    * @param size
119    *             Die Dimension des Problems.
120    * @param omega
121    *             Der Daempfungsparameter.
122    * @param gamma
123    *             Der Parameter fuer eine ueber-/Unterrelaxation der
124    *             Hauptdiagonaleintraege der ILU-Zerlegung.
125    * @return Die approximierte Loesung des linearen Gleichungssystem.
126    */
```

```
127     double *richardson(Matrix *m, double x0[], double e,
128                        double b[], int size, double omega, double gamma);
129
130 public:
131     /**
132      * Erzeugt eine neue Instanz der Klasse <code>WorkingThread</code> mit
133      * den gegebenen Parametern.
134      *
135      * @param epsilon
136      *             Der Epsilon-Faktor innerhalb der Gleichung.
137      * @param beta1
138      *             Die erste Komponente des Beta-Vektors innerhalb der
139      *             Gleichung.
140      * @param beta2
141      *             Die zweite Komponente des Beta-Vektors innerhalb der
142      *             Gleichung.
143      * @param e
144      *             Die Fehlerschranke fuer die Berechnung.
145      * @param n
146      *             Die Aufloesung der Diskretisierungsgitters.
147      * @param omega
148      *             Der Daempfungsparameter fuer die Richardson-Iteration.
149      * @param gamma
150      *             Der Parameter fuer eine ueber-/Unterrelaxation der
151      *             Hauptdiagonaleintraege der ILU-Zerlegung.
152      * @param fileName3DPlot
153      *             Der Dateiname, unter dem die dreidimensionale
154      *             Visualisierung gespeichert werden soll. Ist dies ein
155      *             leerer String, so wird diese Visualisierung nicht
156      *             geplottet. Die Datei wird als pdf gespeichert.
157      * @param useUpwinding
158      *             Bestimmt ob die Upwind-Diskretisierung verwendet werden
159      *             soll oder nicht.
160      */
161     Workthread(double epsilon, double beta1, double beta2,
162                double e, int n, double omega, double gamma,
163                QString fileName3DPlot, bool useUpwinding);
164     /**
165      * Der Destruktor.
166      */
167     ~Workthread();
168
169     virtual void run();
170
171     /**
172      * Diese Methode stoppt die Arbeit dieser Instanz und gibt allen
173      * Speicher wieder frei.
174      */
175     void stop();
176
177     /**
178      * Diese Methode pausiert die Arbeit des Threads. Falls er schon
```

```
179     * pausiert wurde, so macht diese Methode nichts.
180     */
181    void pause();
182
183    /**
184     * Diese Methode setzt die Arbeit des Threads fort. Falls er noch
185     * nicht pausiert wurde, so macht diese Methode nichts.
186     */
187    void resume();
188
189    /**
190     * Stellt fest, ob der Thread pausiert wurde.
191     *
192     * @returns true Genau dann wenn der Thread zur Zeit noch pausiert ist.
193     */
194    bool isPaused();
195
196 signals:
197    /**
198     * Dieses Signal wird ausgeloest, wenn eine neue Genauigkeit berechnet
199     * wurde und zur Anzeige bereitsteht.
200     *
201     * @param acc
202     *              Die neue Genauigkeit.
203     */
204    void newAccuracy(QString acc);
205
206    /**
207     * Dieses Signal wird ausgeloest, wenn ein weiterer Iterationsschritt
208     * berechnet wurde und zur Anzeige bereitsteht.
209     *
210     * @param steps
211     *               Die neue Anzahl aktueller Iterationsschritte.
212     */
213    void newIterationSteps(QString steps);
214
215    /**
216     * Dieses Signal wird ausgeloest, wenn eine neue Nachricht zum Loggen
217     * bereitsteht.
218     *
219     * @param str
220     *              Die zu loggende Nachricht.
221     */
222    void newLogMessage(QString str);
223
224    /**
225     * Dieses Signal wird ausgeloest, wenn eine neue Visualisierung zur
226     * Darstellung bereitsteht.
227     *
228     * @param img
229     *              Das zu zeichnende Bild.
230     */
```

```
231     void new2DGraphic(QImage img);
232 };
233
234 /**
235  * Diese Funktion berechnet die Maximumsnorm fuer die Defektkorrektur.
236  *
237  * @param temp
238  *              Der aktuelle Ergebnisvektor.
239  * @param m
240  *              Die Koeffizientenmatrix des LGS.
241  * @param b
242  *              Die rechte Seite des LGS
243  * @param size
244  *              Die Groesse der Vektoren.
245  * @returns
246  *              Die Maximumsnorm fuer die Defektkorrektur.
247  */
248 double calcNorm(double *temp, Matrix *m, double *b, int size);
249
250 /**
251  * Diese Funktion liefert einen Zeiger auf eine Kopie des gegebenen
252  * Vektors zurueck.
253  *
254  * @param arr
255  *              Der zu kopierende Vektor.
256  * @param size
257  *              Die Groesse des Vektors.
258  * @returns
259  *              Eine Kopie des Vektors.
260  */
261 double *copy(double *arr, int size);
262
263 /**
264  * Diese Funktion liefert das minimale Element in dem gegebenen Array
265  * zurueck.
266  *
267  * @param arr
268  *              Der zu durchsuchende Vektor.
269  * @param size
270  *              Die Groesse des Vektors.
271  * @returns
272  *              Das minimale Element in dem gegebenen Vektor.
273  */
274 double min(double *arr, int size);
275
276 /**
277  * Diese Funktion liefert das maximale Element in dem gegebenen Array
278  * zurueck.
279  *
280  * @param arr
281  *              Der zu durchsuchende Vektor.
282  * @param size
```

```
283  *              Die Groesse des Vektors.
284  * @returns
285  *              Das maximale Element in dem gegebenen Vektor.
286  */
287 double max(double *arr, int size);
288
289 /**
290  * Diese Methode plottet den gegebenen Ergebnisvektor mithilfe von gnuplot
291  * und speichert die Datei unter dem angegebenen Namen.
292  *
293  * @param filename
294  *                  Der zu benutzende Dateiname fuer das Diagramm.
295  * @param result
296  *                  Der Ergebnisvektor ohne Randbedingungen.
297  * @param n
298  *                  Die Aufloesung des Diskretisierungsgitters.
299  * @returns
300  *                  "true" gdw. gnuplot die Visualisierung erfolgreich
301  *                  geplottet hat.
302  */
303 bool saveAs3DDiagram(QString filename, double *result, int n);
304
305 #endif // WORKTHREAD_H
```

Listing A.7: workthread.h

```
1  #include "workthread.h"
2
3  #include <QImage>
4  #include <QTimer>
5  #include <QtConcurrentRun>
6  #include <cmath>
7  #include <QTemporaryFile>
8  #include <QProcess>
9
10 Workthread::Workthread(double epsilon, double beta1, double beta2,
11                        double e, int n, double omega, double gamma,
12                        QString fileName3DPlot, bool useUpwinding) {
13   this->epsilon = epsilon;
14   this->beta1 = beta1;
15   this->beta2 = beta2;
16   this->e = e;
17   this->n = n;
18   this->pixmap = pixmap;
19   this->omega = omega;
20   this->gamma = gamma;
21   this->stopped = false;
22   this->paused = false;
23   this->fileName3DPlot = fileName3DPlot;
24   this->useUpwinding = useUpwinding;
25 }
26
27 Workthread::~Workthread() {
```

```
28 }
29
30 void Workthread::stop() {
31   /* Falls der Thread noch pausiert ist, müssen wir ihn erst fortfahren
32      lassen, bevor er gestopt werden kann. */
33   mutex.lock();
34   stopped = true;
35   paused = false;
36   wcondition.wakeAll();
37   mutex.unlock();
38 }
39
40 void Workthread::pause() {
41   mutex.lock();
42   if (!paused) {
43     paused = true;
44     emit newLogMessage("Berechnung wurde unterbrochen");
45   }
46   mutex.unlock();
47 }
48
49 void Workthread::resume() {
50   mutex.lock();
51   if (paused) {
52     /* Wecke den Thread wieder auf. */
53     paused = false;
54     wcondition.wakeAll();
55     emit newLogMessage("Berechnung wurde fortgesetzt");
56   }
57   mutex.unlock();
58 }
59
60 bool Workthread::isPaused() {
61   return paused;
62 }
63
64 double calcNorm(double *temp, Matrix *m, double *b, int size) {
65   /* ||Ax - b|| */
66   m->multiply(temp);
67   Matrix::sub(temp, b, size);
68   double norm = Matrix::infinityNorm(temp, size);
69
70   /* Aufräumen und Ergebnis zurückliefern. */
71   delete[] temp;
72
73   return norm;
74 }
75
76 double *Workthread::richardson(Matrix *m, double x0[], double e,
77                                double b[], int size, double omega,
78                                double gamma) {
79   /* Berechne zunaechst die ILU-Zerlegung. */
```

```
80    emit newLogMessage("Berechne ILU-Zerlegung");
81    Matrix *ILUM = m->getILU();
82    ILUM->addRelaxation(gamma);
83    emit newLogMessage("ILU-Zerlegung berechnet");
84
85    /* Modifiziere die rechte Seite des LGS. */
86    double *btilde = copy(b, size);
87    ILUM->multiplyWithInverseAsILU(btilde);
88    Matrix::multiply(omega, btilde, size);
89
90    /* Einige Vorbereitungen. */
91    double *xn = copy(x0, size);
92    double norm = 0;
93    int iterCount = 0;
94
95    /* Beginne hier die Iteration. */
96    emit newLogMessage("Beginne Iteration");
97    QFuture<double> futureNorm;
98    bool firstRun = true;
99    do {
100     /* x = x - (w(LU)^-1 x - w(LU)^-1 b) */
101     double *temp = copy(xn, size);
102     m->multiply(temp);
103     ILUM->multiplyWithInverseAsILU(temp);
104     Matrix::multiply(omega, temp, size);
105     Matrix::sub(temp, btilde, size);
106
107     /* Bevor wir den finalen Schritt in der Iteration machen, pruefen wir,
108        ob wir die notwendige Genauigkeit bereits erreicht haben (Dies ist
109        wegen der asynchronen Berechnung der Norm des Vektors
110        notwendig). */
111     if (!firstRun) {
112       norm = futureNorm.result();
113       /* Falls das Resultat genau genug ist, verlassen wir die Funktion
114          und rechnen nicht weiter. */
115       if (norm < e) {
116         delete[] temp;
117         break;
118       }
119     } else {
120       firstRun = false;
121     }
122
123     Matrix::sub(xn, temp, size);
124     delete[] temp;
125
126     /* Berechne die Genauigkeit asynchron. */
127     temp = copy(xn, size);
128     futureNorm = QtConcurrent::run(calcNorm, temp, m, b, size);
129
130     iterCount++;
131
```

```
132     /* Informiere den Benutzer ueber die aktuelle Genauigkeit und die
133        aktuelle Anzahl von Iterationsschritten. */
134     emit newAccuracy(QString::number(norm));
135     emit newIterationSteps(QString::number(iterCount));
136
137
138     /* Falls wir pausiert wurden, halte hier an. */
139     while (paused) {
140       mutex.lock();
141       wcondition.wait(&mutex);
142       mutex.unlock();
143     }
144   } while (!stopped);
145
146   /* Stell sicher, dass die asynchrone Berechnung des Residuums beendet
147      wird. Wenn das nicht gemacht wird, kann das Programm u.U. bei
148      fehlerhaften Koeffizienten abstürzen. */
149   futureNorm.result();
150
151   /* Je nachdem ob der Thread gestoppt wurde oder nicht, geben wir die
152      richtige Nachricht aus. */
153   if (!stopped) {
154     emit newLogMessage("Iteration beendet");
155   } else {
156     emit newLogMessage("Iteration gestoppt");
157   }
158
159   /* Zusammenfassung anzeigen. */
160   emit newLogMessage("Benötigte Schritte: " +
161                      QString::number(iterCount));
162   emit newLogMessage("Genauigkeit: " + QString::number(norm));
163   emit newAccuracy("n/a");
164   emit newIterationSteps("n/a");
165
166   /* Gib alles wieder frei, was wir benoetigt haben. */
167   delete[] btilde;
168   delete ILUM;
169
170   return xn;
171 }
172
173 void Workthread::run() {
174   int size = (n - 1) * (n - 1);
175
176   /* Bereite alles fuer die Eingabe vor. */
177   double *f = new double[size];
178   double *x0 = new double[size];
179   for (int i = 0; i < size; i++) {
180     f[i] = 1.0;
181     x0[i] = 0.0;
182   }
183
```

```
184   /* Hole die Matrix und iteriere. */
185   Matrix *m = Matrix::getMatrix(n, beta1, beta2, epsilon, useUpwinding);
186   double *res = richardson(m, x0, e, f, size, omega, gamma);
187
188   /* Das meiste benoetigen wir nun nach der Iteration nicht mehr. */
189   delete [] f;
190   delete [] x0;
191   delete m;
192
193   /* Falls wir vorzeitig gestoppt wurden, breche hier ab. */
194   if (stopped) {
195     delete [] res;
196     return;
197   }
198
199   /* Ansonsten werden wir hier nun die graphische Darstellung
200      berechnen. */
201   QImage img(n+1, n+1, QImage::Format_RGB32);
202
203   /* Finde den minimalen und maximalen Wert in dem Ergebnisvektor
204      (Asynchron) */
205   QFuture<double> fmin = QtConcurrent::run(min, res, size);
206   QFuture<double> fmax = QtConcurrent::run(max, res, size);
207
208   /* Verwende min und max fuer einen linearen Farbverlauf. */
209   int k = 0;
210   double min = fmin.result();
211   double max = fmax.result();
212   for (int x = 0; x <= n; x++) {
213     for (int y = 0; y <= n; y++) {
214       double val;
215       /* Der Rand ist stets 0. */
216       if (x == 0 || y == 0 || x == n || y == n) {
217         val = 0.0;
218       } else {
219         val = res[k];
220         k++;
221       }
222       int color = (int) (255.0 / (max - min) * val - (255.0 * min)
223                          / (max - min));
224       int colVal = color;
225       /* Farbe berechnen und Pixel setzen. */
226       img.setPixel(x, y, 256 * (256 * colVal + colVal) + colVal);
227     }
228   }
229
230   if (!fileName3DPlot.isEmpty()) {
231     emit newLogMessage("Plotte dreidimensionale Visualisierung");
232     if (!saveAs3DDiagram(fileName3DPlot, res, n)) {
233       emit newLogMessage("Plotten fehlgeschlagen");
234     }
235   }
```

```
236
237   /* Der Ergebnisvektor wird nicht laenger gebraucht. */
238   delete[] res;
239
240   /* Zeige die Daten an. */
241   emit new2DGraphic(img);
242 }
243
244 bool saveAs3DDiagram(QString filename, double *result, int n) {
245
246   /* Bilde den Ergebnisvektor zunächst wieder auf ein zweidimensionales
247      Array ab. */
248   double arr[n + 1][n + 1];
249   int k = 0;
250   for (int x = 0; x <= n; x++) {
251     for (int y = 0; y <= n; y++) {
252       /* Der Rand ist überall 0. */
253       if (x == 0 || y == 0 || x == n || y == n) {
254         arr[x][y] = 0;
255       } else {
256         arr[x][y] = result[k];
257         k++;
258       }
259     }
260   }
261
262   /* Bring die Daten in ein für Gnuplot lesbares Format. */
263   QTemporaryFile *file = new QTemporaryFile();
264   file->open();
265   QString str("");
266   double h = 1.0 / n;
267   for (int y = 0; y <= n; y++) {
268     for (int x = 0; x <= n; x++) {
269       str += QString::number(x * h) + "\t" + QString::number(y * h) +
270              "\t" + QString::number(arr[x][y]) + "\n";
271     }
272     str += "\n";
273   }
274   /* Speichere die Daten in einer temporären Datei. */
275   file->write(str.toAscii());
276   file->close();
277
278   /* Versuche Gnuplot zu starten. */
279   QProcess proc(0);
280   proc.start("gnuplot");
281
282   /* Stell sicher, dass der Prozess erfolgreich gestartet wurde! */
283   proc.waitForStarted();
284   if (proc.state() == QProcess::NotRunning) {
285     return false;
286   }
287
```

```
288     /* Das Starten war erfolgreich. Steuere Gnuplot an und übergebe alle
289        Parameter. */
290     str = "set output \"" + filename + "\"\n";
291     proc.write(str.toAscii());
292     proc.write("set terminal pdf\n");
293     proc.write("unset key\n");
294     proc.write("set isosamples 40\n");
295     proc.write("set ticslevel 0\n");
296     proc.write("set xlabel 'x'\n");
297     proc.write("set ylabel 'y'\n");
298     proc.write("set zlabel 'u'\n");
299     str = "splot \'" + file->fileName() + "\' with lines lc -1\n";
300     proc.write(str.toAscii());
301     proc.closeWriteChannel();
302     proc.waitForFinished();
303
304     /* Lösche die temporäre Datei wieder. */
305     delete file;
306
307     return (proc.exitCode() == QProcess::NormalExit);
308 }
309
310 double *copy(double *arr, int size) {
311     /* Alloziiere einen neuen Speicherbereich für die Kopie des Arrays. */
312     double *cp = new double[size];
313
314     /* Kopiere das Array komponentenweise. */
315     for (int i = 0; i < size; i++) {
316       cp[i] = arr[i];
317     }
318
319     return cp;
320 }
321
322 double max(double *arr, int size) {
323     double max = arr[0];
324
325     for (int i = 0; i < size; i++) {
326       if (arr[i] > max) {
327         max = arr[i];
328       }
329     }
330
331     return max;
332 }
333
334 double min(double *arr, int size) {
335     double min = arr[0];
336
337     for (int i = 0; i < size; i++) {
338       if (arr[i] < min) {
339         min = arr[i];
```

```
340       }
341     }
342
343     return min;
344 }
```

Listing A.8: workthread.cpp

```
 1 #ifndef ABOUTDIALOG_H
 2 #define ABOUTDIALOG_H
 3
 4 #include <QDialog>
 5
 6 namespace Ui {
 7   class AboutDialog;
 8 }
 9
10 /**
11  * @brief Ein Fenster, das Informationen ueber das Programm anzeigt.
12  *
13  * Dies ist das About-Fenster der Anwendung. Dieses stellt dem Benutzer
14  * einige Informationen zur Verfuegung.
15  *
16  * @author Nils Christian Ehmke
17  */
18
19 class AboutDialog : public QDialog {
20   Q_OBJECT
21
22 public:
23   /**
24    * Erzeugt eine neue Instanz dieser Klasse.
25    *
26    * @param parent
27    *                 Der Parent dieses Fensters.
28    */
29   explicit AboutDialog(QWidget *parent = 0);
30
31   /**
32      * Der Destruktor fuer diese Klasse.
33      */
34   ~AboutDialog();
35
36 private:
37   /**
38      * Dieses Feld beinhaltet die GUI-Komponenten.
39      */
40   Ui::AboutDialog *ui;
41 };
42
43 #endif // ABOUTDIALOG_H
```

Listing A.9: aboutdialog.h

```
#include "aboutdialog.h"
#include "ui_aboutdialog.h"

AboutDialog::AboutDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::AboutDialog) {
    ui->setupUi(this);
}

AboutDialog::~AboutDialog() {
    delete ui;
}
```

Listing A.10: aboutdialog.cpp

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>AboutDialog</class>
 <widget class="QDialog" name="AboutDialog">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>217</height>
   </rect>
  </property>
  <property name="sizePolicy">
   <sizepolicy hsizetype="Preferred" vsizetype="Preferred">
    <horstretch>0</horstretch>
    <verstretch>0</verstretch>
   </sizepolicy>
  </property>
  <property name="windowTitle">
   <string>Über...</string>
  </property>
  <property name="modal">
   <bool>true</bool>
  </property>
  <layout class="QGridLayout" name="gridLayout">
   <item row="0" column="0">
    <widget class="QLabel" name="label">
     <property name="text">
      <string>Dieses Programm wurde im Rahmen der Bachelorarbeit &amp;quot;Die
          inkomplette LR-Zerlegung für Konvektions-Diffusionsgleichungen&
         quot; entwickelt.</string>
     </property>
     <property name="wordWrap">
      <bool>true</bool>
     </property>
    </widget>
   </item>
```

```
36      <item row="2" column="0">
37       <widget class="QDialogButtonBox" name="buttonBox">
38        <property name="orientation">
39         <enum>Qt::Horizontal</enum>
40        </property>
41        <property name="standardButtons">
42         <set>QDialogButtonBox::Ok</set>
43        </property>
44        <property name="centerButtons">
45         <bool>true</bool>
46        </property>
47       </widget>
48      </item>
49      <item row="1" column="0">
50       <widget class="QLabel" name="label_2">
51        <property name="text">
52         <string>Entwickelt 2011 von Nils Christian Ehmke
53
54 Betreuer: Prof. Dr. Malte Braack
55 Christian-Albrechts-Universität zu Kiel</string>
56        </property>
57       </widget>
58      </item>
59     </layout>
60    </widget>
61    <resources/>
62    <connections>
63     <connection>
64      <sender>buttonBox</sender>
65      <signal>accepted()</signal>
66      <receiver>AboutDialog</receiver>
67      <slot>accept()</slot>
68      <hints>
69       <hint type="sourcelabel">
70        <x>248</x>
71        <y>254</y>
72       </hint>
73       <hint type="destinationlabel">
74        <x>157</x>
75        <y>274</y>
76       </hint>
77      </hints>
78     </connection>
79     <connection>
80      <sender>buttonBox</sender>
81      <signal>rejected()</signal>
82      <receiver>AboutDialog</receiver>
83      <slot>reject()</slot>
84      <hints>
85       <hint type="sourcelabel">
86        <x>316</x>
87        <y>260</y>
```

```
88        </hint>
89        <hint type="destinationlabel">
90         <x>286</x>
91         <y>274</y>
92        </hint>
93       </hints>
94      </connection>
95     </connections>
96    </ui>
```

Listing A.11: aboutdialog.ui

B. Messdaten

e	1. Konstellation	2. Konstellation	3. Konstellation
0.125	762	401	176
0.0625	967	463	181
0.03125	1173	522	186
0.015625	1379	579	190
0.0078125	1585	635	193
0.00390625	1791	691	197
0.001953125	1997	745	200
0.0009765625	2203	799	203
0.00048828125	2408	853	206
0.000244140625	2614	906	209
0.0001220703125	2820	959	212
0.00006103515625	3026	1012	214

Tabelle B.1.: Messwerte der ersten Auswertung

n	1. Konstellation	2. Konstellation	3. Konstellation
4	6	4	-
8	21	10	-
16	76	31	-
32	297	109	28
64	1181	409	96
128	4716	1582	423
256	18855	6222	1764
512	75413	24679	7196

Tabelle B.2.: Messwerte der zweiten Auswertung

ϵ	Iterationen
1	17513
0.1	5538
0.01	374
0.001	84
0.0001	∞
0.00001	∞

Tabelle B.3.: Messwerte der dritten Auswertung

Iterationsschritt	Residuum
1	1
251	0.999916
501	0.989897
751	0.94628
1001	0.873853
1251	0.7886
1501	0.701827
1751	0.619458
2001	0.544053
2251	0.47641
2501	0.416388
2751	0.363479
3001	0.317039
3251	0.276381
3501	0.240845
3751	0.209812
4001	0.182737
4251	0.159131
4501	0.138555
4751	0.120626
5001	0.105008
5251	0.091405
5501	0.0795599
5751	0.0692466

Tabelle B.4.: Messwerte der dritten Auswertung, $\epsilon = 1.0$

Iterationsschritt	Residuum
1	1
251	0.999913
501	0.988856
751	0.934459
1001	0.828651
1251	0.685792
1501	0.530186
1751	0.385652
2001	0.266636
2251	0.17703
2501	0.11385
2751	0.0714143
3001	0.0439394
3251	0.0266215
3501	0.0159361
3751	0.00945063
4001	0.00556198
4251	0.00325373
4501	0.00189418
4751	0.00109843
5001	0.000635013
5251	0.00036618
5501	0.000210727
5751	0.000121067

Tabelle B.5.: Messwerte der dritten Auswertung, $\epsilon = 0.1$

Iterationsschritt	Residuum
1	1
11	1
21	1
31	1
41	1
51	1
61	1
71	1
81	1
91	1
101	1
111	1
121	0.999998
131	0.999987
141	0.999917
151	0.999592
161	0.998344
171	0.994386
181	0.984022
191	0.960871
201	0.917266
211	0.846788
221	0.747334
231	0.624951
241	0.492082
251	0.36428
261	0.252522
271	0.164114
281	0.100362
291	0.0580476
301	0.0317422
311	0.0164668
321	0.00813149
331	0.00383475
341	0.00173238
351	0.00075189
361	0.00031436
371	0.000126923

Tabelle B.6.: Messwerte der dritten Auswertung, $\epsilon = 0.01$

Iterationsschritt	Residuum
1	1.06624
3	1.03036
5	1.0258
7	1.02485
9	1.02463
11	1.02456
13	1.02453
15	1.02452
17	1.02452
19	1.02452
21	1.02452
23	1.02452
25	1.02452
27	1.02452
29	1.02452
31	1.02452
33	1.02452
35	1.02451
37	1.0245
39	1.02443
41	1.02416
43	1.02329
45	1.02083
47	1.01468
49	1.00107
51	0.974409
53	0.928022
55	0.856198
57	0.757039
59	0.634785
61	0.499986
63	0.366908
65	0.249164
67	0.155725
69	0.0891774
71	0.0466249
73	0.0221902
75	0.00958943
77	0.00375452
79	0.00132914
81	0.000424621
83	0.000122186

Tabelle B.7.: Messwerte der dritten Auswertung, $\epsilon = 0.001$

Iterationsschritt	Residuum für $\omega = 1.0$	Residuum für $\omega = 1.4$	Residuum für $\omega = 1.5$	Residuum für $\omega = 1.6$
64	1	1	1	1
128	1	1	0.999996	0.999992
256	0.999899	0.999531	0.997727	0.996598
512	0.988001	0.972247	0.933945	0.917465
1024	0.835304	0.73579	0.572658	0.518751
2048	0.325938	0.187024	0.0721047	0.0513122

Tabelle B.8.: Messwerte des gedämpften Richardson-Verfahrens

Iterationsschritt	Residuum für $\gamma = 0.0$	Residuum für $\gamma = 0.1$	Residuum für $\gamma = -0.1$	Residuum für $\gamma = -0.2$
64	1	1	1	1
128	1	1	1	0.999998
256	0.999895	0.999971	0.99961	0.998518
512	0.987883	0.99397	0.975316	0.948837
1024	0.834853	0.88825	0.753618	0.629806
2048	0.325529	0.43943	0.207147	0.102065

Tabelle B.9.: Messwerte der Über-/Unterrelaxation

Literaturverzeichnis

[Bad01] Bader, M.: *Robuste, parallele Mehrgitterverfahren für die Konvektions-Diffusions-Gleichung.* Institut für Informatik - Lehrstuhl für numerische Programmierung und Ingenieuranwendungen in der Informatik, Technische Universität München, Diss., Januar 2001. http://www5.in.tum.de/pub/bader_diss.pdf

[Beb11] Bebendorf, M.: *Programmierpraktikum numerische Algorithmen (P2E1) (Numerische Lösung der Wärmeleitungsgleichung), Blatt 3.* http://bebendorf.ins.uni-bonn.de/teaching/Praktikum_SS11/blatt03.pdf. Version: 2011

[Bra92] Braess, D.: *Finite Elemente.* 4. Auflage. Springer, 1992

[Bra05] Brand, C.: *Numerische Mathematik I.* http://institute.unileoben.ac.at/amat/lehrbetrieb/num/vl-skript/skripts05/skripts05.html. Version: 2005. – Vorlesungsskriptum

[Bra09] Braack, M.: *Finite Elemente.* http://www.numerik.uni-kiel.de/~mabr/lehre/skripte/fem-braack.pdf. Version: 2009. – Vorlesungsskriptum

[DM11] Davis, T. ; MathWorld–A Wolfram Web Resource: Sparse-Matrix. (27.07.2011). http://mathworld.wolfram.com/SparseMatrix.html

[DR06] Dahmen, W. ; Reusken, A.: *Numerik für Ingenieure und Naturwissenschaftler.* 2. Auflage. Springer, 2006

[EGK08] Eck, C. ; Garcke, H. ; Knaber, P.: *Mathematische Modellierung.* 1. Auflage. Springer, 2008

[Enc11] Encyclopædia Britannica: Siméon-Denis Poisson. (11.05.2011). http://www.britannica.com/EBchecked/topic/466561/Simeon-Denis-Poisson

[Hac86] Hackbusch, W.: *Theorie und Numerik elliptischer Differentialgleichungen.* 1. Auflage. Teubner Verlag, 1986

[Hac93] Hackbusch, W.: *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* 2. Auflage. Teubner Verlag, 1993

[HD08] Hohmann, A. ; Deuflhard, P.: *Numerische Mathematik 1.* 4. Auflage. Gruyter, 2008

[Joh98] Johannsen, K.: *Robuste Mehrgitterverfahren für die Konvektions-Diffusions Gleichung mit wirbelbehafteter Konvektion*, Ruprecht-Karls-Universität Heidelberg, Diss., 1998. `sites.google.com/site/klausjohannsen/files/PhD.pdf`

[Mei11] Meister, A.: *Numerik linearer Gleichungssysteme - Eine Einführung in moderne Verfahren*. 4. Auflage. Vieweg+Teubner Verlag, 2011

[MV] Meijerink, J.A. ; Vorst, H.A. van d.: *An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix*. Mathematics Of Computation

[MW06] Munz, C. ; Westermann, T.: *Numerische Behandlung gewöhnlicher und partieller Differentialgleichungen*. 1. Auflage. Springer, 2006

[PKA03] P. Knaber, P ; Angermann, L.: *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. Springer, 2003

[Pol02] Pollul, B.: *Effiziente Lösungsverfahren für Konvektions-Diffusions-Probleme auf Shishkin-Gittern*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, 2002. `http://home.arcor.de/pollul/pub/Bernhard_Pollul.pdf`

[RXY+00] Rung, T. ; Xue, L. ; Yan, J. ; Schatz, M. ; Thiele, F.: *Numerische Methoden der Thermo- und Fluiddynamik*. 2. Auflage. Technische Universität Berlin, 2000 `http://www.cfd.tu-berlin.de/Lehre/tfd_skript/Main.html`

[Sch11] Schenk, O.: Präkonditionierung. (22.06.2011). `http://informatik.unibas.ch/lehre/ss03/algorechnen/online/folien/lekt9.pdf`

[Stü05] Stürzekarn, M.: Proseminar Numerik - Jacobi- und Gauß-Seidel-Verfahren, Jacobi-Relaxationsverfahren. (2005). `http://www.math.uni-hamburg.de/home/hofmann/lehrveranstaltungen/sommer05/prosem/Vortrag5.pdf`

[Sue08] Suess, D.: *Computational Physics - (Lineare Gleichungssysteme)*. `http://magnet.atp.tuwien.ac.at/suess/cp/cp/linear.pdf`. Version: 2008. – Vorlesungsskriptum

[U.T] U.Trottenberg: *Numerische Mathematik III*. `http://www.scai.fraunhofer.de/de/ueber-uns/institutsleitung/prof-dr-ulrich-trottenberg/lehrstuhl-prof-trottenberg/lehre/vorlesung-numerische-mathematik-iii.html`. – Vorlesungsskriptum

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Selbstständigkeitserklärung

Hiermit versichere ich, Nils Christian Ehmke (Matr.-Nr 834645), die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Diese Arbeit wurde bisher keinem anderen Prüfungsamt in gleicher oder ähnlicher Form vorgelegt und auch nicht veröffentlicht.

_____ _____

(Ort, Datum) (Unterschrift)