Abschlussbericht

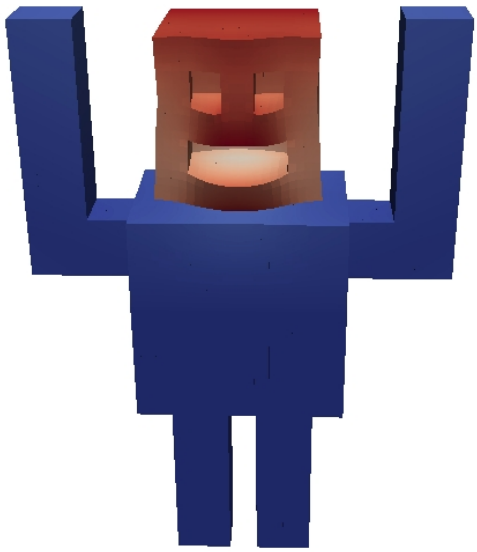
Studienprojekt Modellbildung und Simulation
2011-2012

Lam$\acute{e}$ solver

Studiengang Technomathematik, Bachelor
Lehrstuhl für Angewandte Mathematik und Numerik (LS3)
Fakultät für Mathematik, TU Dortmund

Niklas Borg, Matthias Cebulla, Constantin Christof, Mathias Deska, Jonas Greif, Tim Gutknecht,
Anna Rörich, Sarah Rörich, Stefan Wahlers, Patrick Westervoß

Betreuer: Jun.-Prof. Dr. Dominik Göddeke



Inhaltsverzeichnis

Kapitel 1

Einleitung und Motivation

Im Rahmen des Studienprojekts Modellbildung und Simulation haben wir uns ein Jahr lang mit der Entwicklung einer Simulationsumgebung für komplexe Szenarien der linearisierten Elastizität im $\mathbb{R}^3$ beschäftigt. Dieses Projekt bildete im Verlauf unseres Bachelor-Studiums einen Höhepunkt, da es sowohl Aspekte aus der Mechanik und damit unseres Nebenfachs als auch der angewandten Mathematik miteinander verband. Wir mussten zudem unsere Fähigkeiten im Bereich des selbstständigen wissenschaftlichen Arbeitens unter Beweis stellen. Auch unsere außerfachlichen Kompetenzen waren gefordert. Da es sich bei dem Großteil des Studienprojekts um Teamarbeit handelte, kam auch dieser Aspekt nicht zu kurz und wir lernten, dass die Koordination der Teilgruppen nicht immer einfach ist. Auf diese Weise erhielten wir einen spannenden Einblick in das wissenschaftliche und fachübergreifende Arbeiten.
Im Fachgebiet der Mechanik beschäftigten wir uns mit der Elastizitätslehre, in der man das Deformationsverhalten von Festkörpern unter der Einwirkung äußerer Kräfte untersucht. Diese Problemstellung haben wir als thematische Grundlage für das Studienprojekt aufgrund ihrer vielfältigen Anwendungen ausgewählt, um zum Beispiel die Stabilität von Bauwerken unter Krafteinwirkung beurteilen zu können. Es ist offensichtlich, dass die Verformung von Festkörpern sowohl von der Stärke der Belastung und der Lagerung des Körpers als auch von den Materialeigenschaften abhängig ist. Diese Materialeigenschaften fließen als Parameter in die das Problem beschreibende Gleichung ein, was es uns ermöglicht hat, in dieser Hinsicht flexibel zu bleiben. Bei der Diskretisierung haben wir uns auf finite Differenzen zweiter Ordnung festgelegt und als freien Parameter die Gitterweite h zugelassen. Das Rechnen auf komplexen Geometrien haben wir durch die Verwendung verschiedener Immersed-Boundary-Methoden ermöglicht. Zum Lösen der resultierenden dünn besetzten linearen Gleichungssysteme haben wir unser Softwarepaket um problemspezifische Löser und Vorkonditionierer erweitert. Das effiziente und schnelle Lösen des Gleichungssystems stand nicht nur bei der Wahl der Vorkonditionierer und Löser im Vordergrund, sondern auch bei der Entscheidung nicht nur auf der CPU zu rechnen und den Programmcode dort zu parallelisieren, sondern durch Mitgabe eines Parameters auch das Rechnen auf der GPU zu ermöglichen. In allen Bereichen war uns eine hohe Flexibilität und Modularität der Programme sehr wichtig.

1.1 Struktur des Berichts

Um einen kurzen Überblick über den Bericht zu geben, gehen wir zuerst auf die benötigte Theorie ein und beschäftigen uns dazu zunächst mit dem Thema der linearisierten Elastizität (vgl. Abschnitt 2.1), wobei wir die Lamé-Gleichung, die das obige Problem beschreibt, herleiten. Da wir das Modell der linearisierten Elastizität betrachten, muss sichergestellt werden, dass die Deformationen hinreichend „klein“ und reversibel sind.
Zur Diskretisierung der in dieser Gleichung auftretenden zweiten Ableitungen nutzen wir die Metho-

de der Finiten Differenzen (vgl. Kapitel 2.2) und dort insbesondere die zentralen Differenzenquotienten zweiter Ordnung. Diese Methode lässt sich am Rand des Gebietes offensichtlich nicht anwenden, da hier ins „Leere“ gegriffen werden müsste. Um dies zu vermeiden wird an den Randpunkten auf Vorwärts- beziehungsweise Rückwärts-Differenzenquotienten zweiter Ordnung ausgewichen, wo die Benutzung zentraler Differenzenquotienten nicht möglich ist. Dies kann aufgrund der drei Raumdimensionen, zum Beispiel an den Seitenflächen des betrachteten Körpers, zu Kombinationen beider Differenzenquotienten führen.
Mithilfe dieser Grundüberlegungen beschränken wir die zugrundeliegende Geometrie zunächst auf den Einheitswürfel. Damit wir auch auf komplexeren Geometrien rechnen können, beschäftigen wir uns mit dem Konzept der Immersed-Boundary-Methoden (vgl. Abschnitt 2.3). Hierfür haben wir zwei verschiedene Ansätze studiert und implementiert: die Penalty-Methode aus der Klasse der Continuous-Forcing-Ansätze und aus der Klasse der Discrete-Forcing-Ansätze die Methode der semi-impliziten Integration der Randwerte. So ist es uns möglich weiterhin auf dem Einheitswürfel zu rechnen, jedoch durch Veränderung der Systemmatrix quasi vollkommen beliebige Geometrien „auszuschneiden“.
Anschließend befassen wir uns in dem Abschnitt über Daten- und Löserstrukturen (vgl. Kapitel 2.4) mit der Frage, wie die Struktur der entstehenden Matrix aussieht. Hierbei fällt auf, dass die Matrix je nach Nummerierung der Unbekannten entweder Bandstruktur oder Blockstruktur aufweist, wobei die einzelnen Blöcke wiederum Bandstruktur haben. Um die Effizienz beim Lösen zu erhöhen, schauen wir uns neben den uns bisher bekannten Lösern, dies sind insbesondere Krylov-Unterraum-Verfahren, und Vorkonditionierern auch Blocklöser und -Vorkonditionierer an.
Zum Abschluss des Kapitels geben wir einen Einblick in die Rechnerstruktur von GPUs und Multicore-CPUs. Dabei gehen wir speziell auf die sich aus der spezifischen Rechnerstruktur ergebenden Einsatzmöglichkeiten dieser beiden Hardwarekomponenten ein.
Das darauffolgende Kapitel soll einen Einblick in den Verlauf der Projektarbeit geben und unsere weitere Vorgehensweise im Bezug auf die Implementierung darlegen, wobei wir näher auf die Programmstruktur eingehen. Besonderen Wert haben wir hierbei auf ein modulares Konzept der einzelnen Komponenten gelegt. Auch beim Kompilieren haben wir darauf geachtet, dass die wesentlichen Einstellungen „von außen“ vorgenommen werden können. Zusätzlich wird darauf eingegangen, welche Tests (vgl. Kapitel 3.3) angestrebt und welche Voraussetzungen dafür benötigt werden.
Der Fokus des vierten Kapitels liegt auf den numerischen Tests, die wir unter Benutzung unserer Programme durchgeführt haben. Zunächst demonstrieren wir hier anhand einer analytischen Referenzlösung und dem Hookeschen Gesetz, dass unsere Implementierung Ergebnisse liefert, die denen realer Experimente in guter Näherung entsprechen. Danach gehen wir auf die Frage ein, welche Strategien sich zur Steigerung der Laufzeiteffizienz unter realen Bedingungen eignen und welche Lösungsverfahren praxistauglich sind. Dazu untersuchen wir anhand einiger Testfälle, welcher Speedup sich durch die Nutzung der GPU erreichen lässt und welchen Einfluss die Verwendung der beiden Nummerierungsarten beziehungsweise Speicherformate auf den Lösungsprozess und die Laufzeit hat.
Zuletzt wird anhand einiger komplizierter „Showcases“ die Leistungsfähigkeit und Flexibilität unserer Implementierung demonstriert.
Der Bericht endet mit einem kurzen Fazit, in dem wir die wesentlichen Ergebnisse zusammenfassen.

Kapitel 2

Theoretische Grundlagen

2.1 Linearisierte Elastizität

Elastizität ist ein Teilgebiet der Mechanik, das sich mit der Deformation von Körpern unter der Einwirkung äußerer Kräfte beschäftigt. Diese elastischen Deformationsvorgänge unterliegen in der Realität nichtlinearen Zusammenhängen. Unter gewissen Einschränkungen bezüglich des Materials und der Größe der Verschiebungen stellen die linearisierten Gleichungen jedoch eine gute Approximation zur Beschreibung der Deformation eines Körpers dar. Zusätzlich wollen wir uns auf zeitunabhängige Probleme beschränken. Die Betrachtungen in den nachfolgenden Abschnitten beruhen auf [Wobker] und [Bartel].

2.1.1 Materialeigenschaften

Ein Körper heißt *elastisch*, falls die Deformation des Körpers aufgrund einwirkender Kräfte reversibel ist. Im Folgenden betrachten wir einen festen und deformierbaren Körper, das heißt die einwirkenden Kräfte ändern die Form des Körpers, nicht aber seine Materialeigenschaften. Außerdem beschränken wir uns auf *homogene* und *isotrope* Materialien. Wir nehmen also an, dass jeder Massepunkt in alle Raumrichtungen dieselben Eigenschaften aufweist. Ein solches Material ist beispielsweise Stahl, wobei Holz aufgrund seiner Faserstruktur ein typisches Gegenbeispiel ist.
Wir betrachten also einen Körper $\Omega \subseteq \mathbb{R}^3$ mit Rand $\partial\Omega = \Gamma_D \cup \Gamma_N$ und $\Gamma_D \cap \Gamma_N = \emptyset$ im undeformierten Zustand. Hierbei bezeichnet Γ_D den Dirichlet-Rand und Γ_N den Neumann-Rand, deren Bedeutung im Kapitel 2.1.3 erläutert wird. Wird der Körper mit Volumenkräften (eingeprägten Kräften) f und Oberflächenkräften (Spannungen) t belastet, so verformt er sich. Das gesuchte Verschiebungsfeld $u : \Omega \to \mathbb{R}^3$ beschreibt diese Deformationen.

2.1.2 Lamé-Gleichung

Das Ziel dieses Abschnitts ist es, die dieses Problem beschreibende Gleichung nur in Abhängigkeit der Verschiebungen u darzustellen. Hierzu stehen uns die folgenden aus der Mechanik bekannten Gesetze zur Verfügung:

$$\text{linearisierter Verzerrungstensor:} \quad \varepsilon = \frac{1}{2}(\operatorname{grad}(u)^{\mathsf{T}} + \operatorname{grad}(u)) \tag{2.1}$$

$$\text{Impulsbilanz:} \quad \operatorname{div}(\sigma(x)) = f(x) \quad \forall x \in \Omega \setminus \Gamma \tag{2.2}$$

$$\text{Hookesches Gesetz:} \quad W(\varepsilon) = \mu \operatorname{tr}(\varepsilon^2) + \frac{1}{2}\operatorname{tr}(\varepsilon)^2 \Rightarrow \sigma = 2\mu\varepsilon + \lambda \operatorname{tr}(\varepsilon) I \tag{2.3}$$

Hierbei beschreibt ε die Verzerrung eines Linienelementes aufgrund der Verformung. Durch σ wird der im Körper herrschende Spannungszustand beschrieben, wobei Spannung als Kraft pro Fläche definiert ist:

$$\sigma : \Omega \to \mathbb{R}^{3\times 3}, \quad \sigma = \begin{pmatrix} \sigma_{xx} & \sigma_{yx} & \sigma_{zx} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{zy} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix} \tag{2.4}$$

Bei den Spannungskomponenten handelt es sich um Vektoren, wie sie in Abbildung 2.1 zu sehen sind.

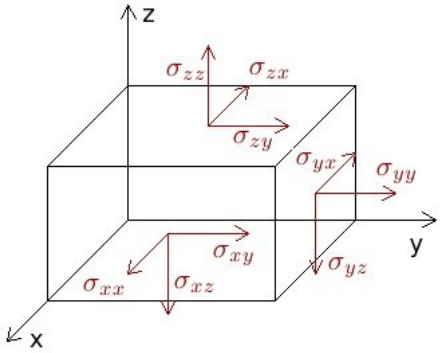


Abbildung 2.1: Dreidimensionaler Spannungszustand

Die im Hookeschen Gesetz auftretenden Größen μ und λ sind Materialkonstanten, die sogenannten Lamé-Konstanten. Sie stehen mit dem Elastizitätsmodul E und der Querkontraktionszahl ν, die experimentell ermittelt werden können, in der Beziehung

$$\mu = \frac{E}{2(1+\nu)}, \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}.$$

Anschaulich kann E als Widerstand gegen die Verformung aufgefasst werden, während ν das negative Verhältnis der relativen Dickenänderung zur relativen Längenänderung angibt.

$$\nu = -\frac{\Delta d/d}{\Delta l/l}$$

Geht $\nu \to 0.5$, so folgt daraus, dass $\lambda \to \infty$ geht und man spricht von fast inkompressiblen Materialien. Im Grenzfall $\nu = 0.5$ heißt das Material dann inkompressibel und das betrachtete Modell der linearisierten Elastizität gilt nicht mehr. Einige Beispiele für die eingeführten Materialkonstanten soll die untenstehende Tabelle 2.1 liefern.

Material	E[Pa]	ν[-]	μ[Pa]	λ[Pa]
Stahl	$2.1 \cdot 10^{11}$	0.29	$8.14 \cdot 10^{10}$	$1.1 \cdot 10^{11}$
Aluminium	$7.0 \cdot 10^{10}$	0.35	$2.6 \cdot 10^{10}$	$6.0 \cdot 10^{10}$
Gummi	$2.5 \cdot 10^{7}$	0.499-0.5	$8.2 \cdot 10^{6}$	$4.1 \cdot 10^{9}$

Tabelle 2.1: Materialkonstanten verschiedener Stoffe [Wobker]

Durch Zusammenführen der Gleichungen (2.1), (2.2) und (2.3) erhalten wir

$$-(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = f \qquad \text{in } \Omega.$$

Um diese Gleichung aus mechanischer und mathematischer Sicht zu vervollständigen, ist es notwendig Randbedingungen vorzuschreiben.

2.1.3 Dirichlet- und Neumann-Randbedingungen

Um die Randbedingungen zu definieren kann man sich überlegen, dass man den Körper, im Rahmen unseres Modells, von außen auf zwei Arten beeinflussen kann. Zum einen kann man ihn durch Auflager in einer bestimmten Position halten. Andererseits ist es möglich Kräfte von außen auf den Körper wirken zu lassen.
Auf dem Dirichlet-Rand Γ_D können Anfangsverschiebungen mittels sogenannter Dirichlet-Randbedingungen

$$u(x) = u_D(x) \qquad \forall x \in \Gamma_D$$

vorgegeben werden. Gilt $u(x) = u_D(x) = 0 \ \ \forall x \in \Gamma_D$, so spricht man von homogenen, anderenfalls von inhomogenen Dirichlet-Randbedingungen. Der Rand des betrachteten Körpers kann sich an diesen Stellen also nicht weiter verformen.
Diese Eigenschaft wird durch die Neumann-Randbedingungen

$$\sigma(x)n = t(x) \qquad \forall x \in \Gamma_N$$

in die Gleichung aufgenommen. Dabei ist n der äußere Normalenvektor im Punkt x. Aus mechanischer Sicht entspricht der Neumann-Rand einem freien Rand.
Mit diesen Randbedingungen können wir nun die *Lamé-Gleichung* aufstellen, die das linear-elastische Verhalten eines belasteten Körpers beschreibt.

$$\begin{aligned} -(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) &= f \quad \text{in } \Omega \\ u &= u_D \quad \text{auf } \Gamma_D \\ \sigma n &= t \quad \text{auf } \Gamma_N \end{aligned} \tag{2.5}$$

2.1.4 Von-Mises-Vergleichsspannungen

Der Spannungstensor (2.4) ist nicht sehr anschaulich und schlecht geeignet, um Materialversagen abzulesen, da kritische Spannungen meist als skalare Größen im Zugversuch ermittelt werden. Hierbei wird das betreffende Material einem einachsigen Spannungszustand ausgesetzt und so die kritische Spannung bestimmt. Wir benötigen daher eine skalare Vergleichsspannung σ_V, die den dreidimensionalen Spannungszustand möglichst gut darstellt. Eine geeignete Größe sind die von-Mises-Spannungen, die über die Formel

$$\sigma_V = \frac{1}{\sqrt{2}}\sqrt{(\sigma_{xx}-\sigma_{yy})^2 + (\sigma_{yy}-\sigma_{zz})^2 + (\sigma_{zz}-\sigma_{xx})^2 + 6({\sigma_{xy}}^2 + {\sigma_{yz}}^2 + {\sigma_{xz}}^2)}$$

und das Hookesche Gesetz (2.3) bestimmt werden (vgl. [Kessel]).

2.2 Diskretisierung mit Finiten Differenzen

Die Methode der *Finiten Differenzen* ist ein Näherungsverfahren zur Diskretisierung von partiellen Differentialgleichungen. In diesem Abschnitt wird auf die Herleitung des Verfahrens über den Satz von Taylor eingegangen, bevor dann die im vorherigen Kapitel eingeführte Lamé-Gleichung (vgl. Gleichung 2.5) diskretisiert wird. Weiterhin wird eine gewisse Grundlage an Theorie gelegt, wobei Begriffe wie Konsistenz, Konvergenz und Stabilität des Verfahrens erläutert werden, um die Qualität der Approximation beurteilen zu können. Schließlich beschäftigt sich dieses Kapitel mit der Diskretisierung von Dirichlet- beziehungsweise Neumann-Randbedingungen.

2.2.1 Approximation zweiter Ableitungen

Grundlage des Verfahrens der Finiten Differenzen ist folgender Satz (vgl. [Königsberger]).

Satz (Taylor)

Sei $I \subset \mathbb{R}$ ein Intervall, sei $u : I \to \mathbb{R}, u \in C^{(n+1)}(I)$.
Dann gilt: $\forall\ a, x \in I :\ u(x) = T_n(x) + R_n(x)$ mit der Taylorentwicklung

$$T_n(x) = \sum_{k=0}^{n} \frac{(x-a)^k}{k!} \left(\frac{\partial^k u}{\partial x^k}\right)(a)$$

um den Entwicklungspunkt a und mit dem Restglied

$$R_n(x) = \int_a^x \frac{(x-t)^n}{n!} \left(\frac{\partial^{n+1} u}{\partial x^{n+1}}\right)(t)dt,$$

das heißt $u(x) = T_n(x) + \mathcal{O}\left((x-a)^{n+1}\right)$.

Die nun folgenden Resultate basieren auf [Möller]. Sei $I = [0,1]$ äquidistant zerlegt mit $x_i = ih$, wobei $i \in \{0, ..., N\},\ N \in \mathbb{N},\ h = \frac{1}{N}$ (vgl. Abb. 2.2) und $u_i := u(x_i),\ \left(\frac{\partial^k u}{\partial x^k}\right)_i := \left(\frac{\partial^k u}{\partial x^k}\right)(x_i)$.

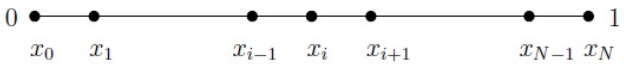


Abbildung 2.2: $[0,1]$ äquidistant in $N+1$ Stützstellen zerlegt

Dann folgt mit dem Satz von Taylor:

$$u_{i+1} = \sum_{k=0}^{n} \frac{h^k}{k!} \left(\frac{\partial^k u}{\partial x^k}\right)_i = u_i + h\left(\frac{\partial u}{\partial x}\right)_i + \frac{h^2}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_i + \frac{h^3}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_i + ... \tag{2.6}$$

$$u_{i-1} = \sum_{k=0}^{n} \frac{(-h)^k}{k!} \left(\frac{\partial^k u}{\partial x^k}\right)_i = u_i - h\left(\frac{\partial u}{\partial x}\right)_i + \frac{h^2}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_i - \frac{h^3}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_i + ... \tag{2.7}$$

Subtrahiert man Gleichung (2.7) von (2.6), so erhält man den zentralen Differenzenquotienten der ersten Ableitung:

$$\left(\frac{\partial u}{\partial x}\right)_i = \frac{u_{i+1} - u_{i-1}}{2h} + \mathcal{O}\left(h^2\right) \tag{2.8}$$

Wenn man die beiden Gleichungen hingegen addiert, so ergibt dies den zentralen Differenzenquotienten der zweiten Ableitung:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \mathcal{O}\left(h^2\right) \tag{2.9}$$

Nun wollen wir die im eindimensionalen gewonnenen Approximationen direkt auf die dritte Dimension erweitern. Betrachten wir also im Folgenden die äquidistante Zerlegung Ω_h von $\Omega := [0,1]^3$ mit Schrittweite $h := \frac{1}{N},\ N \in \mathbb{N}$. Dann hat $(x_i, y_j, z_k) \in \Omega_h$ die Form $(ih, jh, kh),\ i,j,k = 0, ..., N$.
Beispielhaft ist dies in Abbildung 2.3 für $N = 3$ dargestellt.

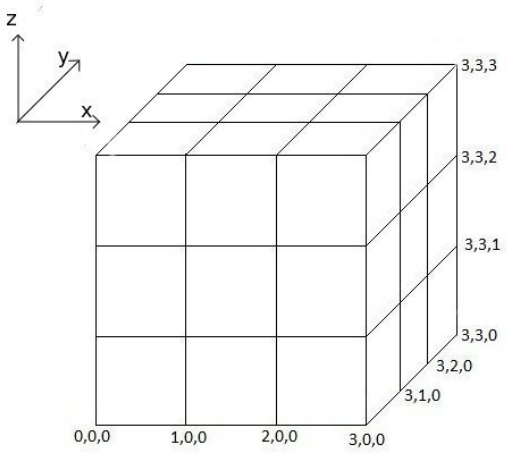


Abbildung 2.3: $[0,1]^3$ äquidistant zerlegt

Wählen wir nun $(x_i, y_j, z_k) \in \Omega_h \backslash \partial\Omega_h$, bilden gemäß Gleichung (2.9) die Näherungen der zweiten Ableitung in x-, y- und z-Richtung und variieren den entsprechenden Index, so erhalten wir für $i,j,k = 1, ..., N-1$:

$$\begin{aligned}
\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j,k} &= \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{h^2} + \mathcal{O}\left(h^2\right) \\
\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j,k} &= \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{h^2} + \mathcal{O}\left(h^2\right) \\
\left(\frac{\partial^2 u}{\partial z^2}\right)_{i,j,k} &= \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{h^2} + \mathcal{O}\left(h^2\right)
\end{aligned}$$

Außerdem kann man gemischte Ableitungen in ähnlicher Weise diskretisieren. So erhält man zum Beispiel eine Approximation von $\left(\frac{\partial^2 u}{\partial x \partial y}\right)$ wie folgt.
Nach Gleichung (2.8) gilt:

$$\left(\frac{\partial u}{\partial x}\right) = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2h} + \mathcal{O}\left(h^2\right) \tag{2.10}$$

Setzt man nun in Gleichung (2.10) die analoge Näherung von $\left(\frac{\partial u}{\partial y}\right)$ ein, so erhält man:

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)_{i,j,k} = \frac{u_{i+1,j+1,k} - u_{i+1,j-1,k} - u_{i-1,j+1,k} + u_{i-1,j-1,k}}{4h^2} + \mathcal{O}\left(h^2\right) \tag{2.11}$$

Analog zu (2.11) können Näherungen für $\left(\frac{\partial^2 u}{\partial x \partial z}\right)_{i,j,k}$ und $\left(\frac{\partial^2 u}{\partial y \partial z}\right)_{i,j,k}$ ermittelt werden.

2.2.2 Konsistenz, Stabilität und Konvergenz

Die Definitionen und Folgerungen in diesem Abschnitt beruhen auf den Darstellungen von [Knabner]. Die Begriffe Konsistenz, Stabilität und Konvergenz werden anhand des eindimensionalen „Modellproblems“, also der Poisson-Gleichung $-\Delta u = f$ mit homogenen Dirichlet-Randbedingungen, untersucht. Die Diskretisierung überführt die Lösung des Modellproblems in die Lösung eines linearen Gleichungssystems der Form $A_h u_h = f_h$. Dabei bezeichnet A_h die aus der Diskretisierung entstehende Matrix, in der die Vorfaktoren der Lösungskomponenten stehen, u_h die angenährte Lösung und f_h die in den Gitterpunkten ausgewertete rechte Seite.
Nun stellt sich natürlich die Frage, wie gut das Näherungsverfahren wirklich ist. Dies lässt sich über den Begriff der Konsistenz beantworten.

Definition (Konsistenz)

Wir betrachten das LGS $A_h u_h = f_h$. Sei $u : \Omega \to \mathbb{R}$ die exakte Lösung der Poisson-Gleichung

$$-\Delta u = f$$

Sei Ω_h die äquidistante Zerlegung von Ω mit Schrittweite h und $u|_{\Omega_h}$ die exakte Lösung ausgewertet an den Gitterpunkten.
Die Approximation von u heißt konsistent, falls gilt:

$$\|A_h\left(u|_{\Omega_h}\right) - A_h u_h\| = \|A_h\left(u|_{\Omega_h}\right) - f_h\| \overset{h\to 0}{\longrightarrow} 0$$

Das Verfahren hat die (Konsistenz-)Ordnung $p > 0$, falls gilt:

$$\|A_h\left(u|_{\Omega_h}\right) - f_h\| = \mathcal{O}\left(h^p\right)$$

Der Konsistenzbegriff liefert eine Aussage darüber, wie gut das Näherungsverfahren das tatsächliche Problem annähert, also in wie weit das diskretisierte Problem mit dem kontinuierlichen übereinstimmt. Dies wird überprüft, indem man den diskretisierten Laplace-Operator nicht nur auf die approximierte Lösung u_h, sondern auch auf die exakte an den Gitterpunkten ausgewertete Lösung $u|_{\Omega_h}$ anwendet und mit dem diskretisierten Laplace-Operator der angenäherten Lösung vergleicht. Je höher dabei die Ordnung ist, desto besser ist die Approximation, da der Fehler $\mathcal{O}\left(h^p\right)$ für wachsendes p immer kleiner wird.

Für den Zusammenhang zwischen Konsistenzfehler und dem Fehler von exakter und approximierter Lösung gilt:

$$\|u_h - u|_{\Omega_h}\| \leq \|A_h^{-1}\| \|A_h u_h - f_h\| \tag{2.12}$$

Um einen proportionalen Zusammenhang zwischen dem Konsistenzfehler und dem „Konvergenzfehler“ $\|u_h - u|_{\Omega_h}\|$ zu erhalten, muss $\|A_h^{-1}\|$ von h unabhängig und beschränkt sein.
Dies führt auf den Begriff der Stabilität.

Definition (Stabilität)

Die Näherungslösung u_h heißt stabil, falls gilt: $\exists\, C > 0$ unabhängig von h, so dass $\|A_h^{-1}\| \leq C$.

An der Abschätzung (2.12) lässt sich ablesen, dass Konsistenz und Stabilität die Konvergenz implizieren und dabei mindestens die Konsistenzordnung als Konvergenzordnung vorliegt.

Satz (Konvergenz)

Konsistenz und Stabilität liefern Konvergenz, das heißt $\|u_h - u|_{\Omega_h}\| \xrightarrow{h\to 0} 0$.
Sei weiter p die Konsistenzordnung und q die Konvergenzordnung.
Dann gilt: $q \geq p$.

Des Weiteren lässt sich zeigen, dass die Konsistenzordnung direkte Auswirkungen auf die Kondition der Systemmatrix A_h beziehungsweise deren Wachstumsverhalten hat. Im Allgemeinen gilt hier, dass die Konditionszahl bei der Verwendung eines Verfahrens der Ordnung p für $h \to 0$ wie h^{-p} gegen unendlich strebt. Details hierzu finden sich etwa bei [Roos].

2.2.3 Diskretisierte Lamé-Gleichung

Die folgenden Darstellungen sind analog zu dem Vorgehen von [Wobker] hergeleitet. Nachdem wir uns im vorangegangenen Teil dieses Kapitels mit der Methode der Finiten Differenzen beschäftigt und uns davon überzeugt haben, dass das Verfahren geeignet ist, um partielle Differentialgleichungen zu diskretisieren, wollen wir es nun auf die Lamé-Gleichung

$$-(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = f$$

konkret anwenden. Mit den Bezeichnungen u_x, u_y und u_z für die Verschiebungen in x-, y- und z-Richtung sowie f_x, f_y und f_z für die Komponenten der rechten Seite lässt sich die Lamé-Gleichung umschreiben in

$$\begin{pmatrix} (2\mu+\lambda)\partial_{xx}u_x & + & \mu(\partial_{yy}u_x + \partial_{zz}u_x) & + & (\mu+\lambda)\partial_{xy}u_y & + & (\mu+\lambda)\partial_{xz}u_z \\ (\mu+\lambda)\partial_{xy}u_x & + & (2\mu+\lambda)\partial_{yy}u_y & + & \mu(\partial_{xx}u_y + \partial_{zz}u_y) & + & (\mu+\lambda)\partial_{yz}u_z \\ (\mu+\lambda)\partial_{xz}u_x & + & (\mu+\lambda)\partial_{yz}u_y & + & (2\mu+\lambda)\partial_{zz}u_z & + & \mu(\partial_{xx}u_z + \partial_{yy}u_z) \end{pmatrix} = -\begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix}.$$

Approximieren wir nun die Ableitungen mit Finiten Differenzen, so erhalten wir die diskretisierte Lamé-Gleichung.

$$\begin{aligned} -f_{x_{i,j,k}}h^2 \;=\; & \underbrace{\begin{aligned}&(2\mu+\lambda)u_{x_{i-1,j,k}} + \mu u_{x_{i,j-1,k}} + \mu u_{x_{i,j,k-1}} + 2(4\mu+\lambda)u_{x_{i,j,k}} \\ &+(2\mu+\lambda)u_{x_{i+1,j,k}} + \mu u_{x_{i,j+1,k}} + \mu u_{x_{i,j,k+1}}\end{aligned}}_{\mathrm{k}_{11}(u_{i,j,k})} \\ & + \underbrace{\frac{\mu+\lambda}{4}(u_{y_{i+1,j+1,k}} - u_{y_{i+1,j-1,k}} - u_{y_{i-1,j+1,k}} + u_{y_{i-1,j-1,k}})}_{\mathrm{k}_{12}(u_{i,j,k})} \\ & + \underbrace{\frac{\mu+\lambda}{4}(u_{z_{i+1,j,k+1}} - u_{z_{i+1,j,k-1}} - u_{z_{i-1,j,k+1}} + u_{z_{i-1,j,k-1}})}_{\mathrm{k}_{13}(u_{i,j,k})} \end{aligned}$$

$$\begin{aligned} -f_{y_{i,j,k}}h^2 \;=\; & \underbrace{\frac{\mu+\lambda}{4}(u_{x_{i+1,j+1,k}} - u_{x_{i+1,j-1,k}} - u_{x_{i-1,j+1,k}} + u_{x_{i-1,j-1,k}})}_{\mathrm{k}_{21}(u_{i,j,k})} \\ & \underbrace{\begin{aligned}&+\mu u_{y_{i-1,j,k}} + (2\mu+\lambda)u_{y_{i,j-1,k}} + \mu u_{y_{i,j,k-1}} + 2(4\mu+\lambda)u_{y_{i,j,k}} \\ &+\mu u_{y_{i+1,j,k}} + (2\mu+\lambda)u_{y_{i,j+1,k}} + \mu u_{y_{i,j,k+1}}\end{aligned}}_{\mathrm{k}_{22}(u_{i,j,k})} \\ & + \underbrace{\frac{\mu+\lambda}{4}(u_{z_{i,j+1,k+1}} - u_{z_{i,j+1,k-1}} - u_{z_{i,j-1,k+1}} + u_{z_{i,j-1,k-1}})}_{\mathrm{k}_{23}(u_{i,j,k})} \end{aligned}$$

$$
\begin{aligned}
-f_{z_{i,j,k}}h^2 \quad &= \underbrace{\frac{\mu+\lambda}{4}(u_{x_{i+1,j,k+1}} - u_{x_{i+1,j,k-1}} - u_{x_{i-1,j,k+1}} + u_{x_{i-1,j,k-1}})}_{\mathrm{k}_{31}(u_{i,j,k})} \\
&\quad + \underbrace{\frac{\mu+\lambda}{4}(u_{y_{i,j+1,k+1}} - u_{y_{i,j+1,k-1}} - u_{y_{i,j-1,k+1}} + u_{y_{i,j-1,k-1}})}_{\mathrm{k}_{32}(u_{i,j,k})} \\
&\quad \underbrace{\begin{aligned}+\mu u_{z_{i-1,j,k}} + \mu u_{z_{i,j-1,k}} + (2\mu+\lambda)u_{z_{i,j,k-1}} + 2(4\mu+\lambda)u_{z_{i,j,k}} \\ +\mu u_{z_{i+1,j,k}} + \mu u_{z_{i,j+1,k}} + (2\mu+\lambda)u_{z_{i,j,k+1}}\end{aligned}}_{\mathrm{k}_{33}(u_{i,j,k})}
\end{aligned} \tag{2.13}
$$

Zur späteren Verwendung und Vereinfachung bezeichnen wir die einzelnen Kopplungsterme als

$$k_{st}(u_{i,j,k}),\ s,t \in \{1,2,3\}\,.$$

2.2.4 Behandlung von Dirichlet- und Neumann-Randbedingungen

Wir betrachten zuerst die zuvor diskretisierte Lamé-Gleichung mit den Dirichlet-Randbedingungen $u = u_D(x,y,z),\ u_D : \partial\Omega \to \mathbb{R}^3$. Sei $N \in \mathbb{N}$ und $h := \frac{1}{N}$. Wähle Ω_h wie in Kapitel 2.2.1 beschrieben. Zur Veranschaulichung wird beispielhaft nur ein Randpunkt mit inhomogener Dirichlet-Randbedingung versehen. Sei exemplarisch

$$u_D(x_i,y_j,z_k) = \begin{cases} u_D := \left(u_{D_x}, u_{D_y}, u_{D_z}\right)^\top & i=1, j=0, k=1 \\ 0, & \text{sonst} \end{cases}$$

wobei $u_D \neq 0$.
Also liegt in diesem Beispiel eine inhomogene Randbedingung im Punkt $(h,0,h)$ vor. Um diese in die Diskretisierung zu integrieren, fügt man an der entsprechenden Stelle in der Systemmatrix Einheitszeilen ein, in denen an den passenden Stellen eine 1 und sonst 0 steht. Diese Stelle ist von der Nummerierung der Unbekannten im Lösungsvektor u abhängig (vgl. Kapitel 2.4). In den Vektor der rechten Seite wird dann an der entsprechenden Stelle der zugehörige Wert der (in)homogenen Randbedingung eingesetzt.

Bei der Betrachtung von Neumann-Randbedingungen ist die Diskretisierung etwas komplizierter. Diese Randbedingungen haben für die Lamé-Gleichung die Form

$$\sigma n = (2\mu\varepsilon + \lambda\,\mathrm{tr}(\varepsilon)I)n = t \tag{2.14}$$

wobei μ und ε die entsprechenden Größen in der Lamé-Gleichung bezeichnen und n der lokale Normalenvektor ist, der demnach orthogonal auf der Oberfläche des Gebietes steht. Hierbei gibt t den Wert der zur Neumann-Randbedingung gehörigen Spannung an. In Matrix-Vektor-Schreibweise ergibt sich

$$\left[2\mu \begin{pmatrix} \frac{\partial u_x}{\partial x} & \frac{1}{2}\left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z}\right) \\ \frac{1}{2}\left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x}\right) & \frac{\partial u_y}{\partial y} & \frac{1}{2}\left(\frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z}\right) & \frac{1}{2}\left(\frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y}\right) & \frac{\partial u_z}{\partial z} \end{pmatrix} + \lambda\left(\frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}\right) I \right] n = t.$$

Um eine Diskretisierung von (2.14) am Neumann-Rand zu erhalten, muss also der Spannungstensor σ approximiert werden. Dabei ist zu beachten, auf welchem Teil des Randes man sich befindet. Betrachtet man zum Beispiel einen Punkt auf einer Seitenfläche, so muss ein einseitiger Differenzenquotient verwendet werden. Analog müssen für Kanten beziehungsweise Ecken bis zu drei einseitige Differenzenquotienten eingesetzt werden. Dies ist nötig, weil man nicht zu Punkten außerhalb des Gebietes koppeln

kann. Um trotzdem Konsistenzordnung zwei zu erhalten, muss man weiter ins Innere greifen. Ähnlich zu Kapitel 2.2.1 kann folgender einseitiger Differenzenquotient gebildet werden

$$\left(\frac{\partial u}{\partial x}\right)_{i,j,k} = \frac{-3u_{i,j,k} + 4u_{i+1,j,k} - u_{i+2,j,k}}{2h} + \mathcal{O}\left(h^2\right).$$

Analog lassen sich Näherungen für $\left(\frac{\partial u}{\partial y}\right)_{i,j,k}$ und $\left(\frac{\partial u}{\partial z}\right)_{i,j,k}$ herleiten.
Setzt man nun diese Approximationen je nach Bedarf in die Matrix-Vektor-Schreibweise von Gleichung (2.14) ein, so erhält man die Diskretisierung zweiter Ordnung der Neumann-Randbedingungen. Dies hat allerdings zur Folge, dass in der Systemmatrix durch die einseitigen Differenzenquotienten zusätzliche Einträge entstehen, je nach Nummerierung des Gitters auch Bänder. Des Weiteren hat die Verwendung von einseitigen Differenzenquotienten den Verlust der Symmetrie der Systemmatrix zur Folge, was die Wahl des Lösers zusätzlich einschränkt (vgl. Kapitel 2.4).

2.3 Immersed-Boundary-Methoden

Die Methoden, die in Kapitel 2.2.4 zur Berücksichtigung von Dirichlet-Randbedingungen vorgestellt wurden, erfordern, dass das Diskretisierungsgitter dieselbe Form hat wie das untersuchte Gebiet. Für eine komplexe Geometrie lässt sich ein solches randadaptiertes Netz jedoch im Allgemeinen nur unter sehr hohem Aufwand generieren. Um die Geometrie einfacher erfassen zu können, kann es hier sinnvoll sein, nicht die Diskretisierung anzupassen, sondern die zu lösende Differentialgleichung. Die hierzu verwendeten Verfahren heißen Immersed-Boundary-Methoden. Anhand der Art und Weise, wie die Manipulationen an den Gleichungen erfolgen, lassen sich Immersed-Boundary-Methoden klassifizieren und grob in zwei Hauptgruppen einteilen [Mittal]. Diese unterscheiden sich primär darin, an welcher Stelle der Rechnung die Randbedingungen in die Diskretisierung aufgenommen werden. Im Folgenden werden wir zwei verschiedene Ansätze, die Penalty-Methode und die direkte Integration der Randbedingungen, am Beispiel der Lamé-Gleichung vorstellen.

2.3.1 Penalty-Methode

Die Penalty-Methode gehört in die Klasse der sogenannten Continuous-Forcing-Verfahren, die auf dem Ansatz basieren, die Randbedingungen durch einen zusätzlichen Lastterm f_R direkt in die untersuchte Differentialgleichung zu integrieren. Stellt diese Ersatzlast sicher, dass die Lösung den Nebenbedingungen einer bestimmten Geometrie genügt, so brauchen diese in der veränderten Differentialgleichung nicht mehr explizit als Restriktionen behandelt werden. Im Falle der Lamé-Gleichung ist diese Herangehendsweise dazu äquivalent, dass die Einspannung an fixierten Punkten durch die (unbekannten) Auflagerkräfte ersetzt wird und diese wiederum als äußere Kräfte interpretiert und auf die rechte Seite der Gleichung geschrieben werden. Das fundamentale Problem dieses Ansatzes besteht darin, dass die passende Ersatzlast im Allgemeinen nicht explizit berechnet werden kann. Das Penalty-Verfahren umgeht diese Problematik, indem es die erforderliche Belastung während der Kalkulation aus einer Art Hookeschem Gesetz bestimmt. Dazu lässt man (im Falle homogener Dirichlet-Randbedingungen) an eingespannten Punkten die zusätzliche Kraft $f_R = -Cu$ für ein festes $C \gg 1$ wirken, die jede Auslenkung aus dem Nullpunkt durch eine Rückstellkraft bestraft. Lösen wir beispielsweise die Lamé-Gleichung auf dem Einheitswürfel $[0,1]^3$ unter der Bedingung $u|_{[0,1]^3\setminus\Omega} = 0, \Omega \subset]0,1[^3$, so erlaubt uns diese Ersatzlast, die Differentialgleichung umzuschreiben zu

$$-(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = f + f_R \mathbb{1}_{[0,1]^3\setminus\Omega} \qquad \text{in } [0,1]^3$$

Hierbei ist $\mathbb{1}_{[0,1]^3\setminus\Omega}$ die charakteristische Funktion auf $[0,1]^3 \setminus \Omega$. Zur Diskretisierung schränkt man nun die Kraft $-Cu\mathbb{1}_{[0,1]^3\setminus\Omega}$ auf eine beliebige Punktmenge $\{x_k\}_{k=1,\dots,n}$ aus dem eingespannten Bereich ein, das heißt man betrachtet

$$-(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = f - \sum_{k=1}^{n} Cu\delta_{x_k} \qquad \text{in } [0,1]^3.$$

Wollen wir diese Differentialgleichung auf einem kartesischen Gitter lösen, so muss nun noch das Kronecker Delta durch ein kontinuierliches Pendant $\widetilde{\delta}$, eine sogenannte Lastdistributionsfunktion, ersetzt werden, damit die Kraftfunktion in den Gitterpunkten ausgewertet werden kann. Einige Beispiele hierzu finden sich in Abbildung 2.4.

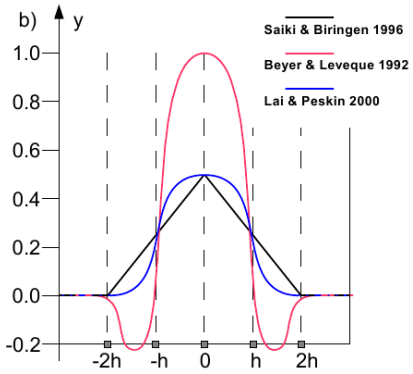


Abbildung 2.4: Verschiedene Lastdistributionsfunktionen nach [Mittal]

Die resultierende Gleichung kann nun analog zu Kapitel 2.2.3 diskretisiert werden. Für die x-Richtung ergibt sich beispielsweise

$$\begin{aligned} -f_{x_{i,j,k}}h^2 &= (2\mu+\lambda)u_{x_{i-1,j,k}} + \mu u_{x_{i,j-1,k}} + \mu u_{x_{i,j,k-1}} + 2(4\mu+\lambda+C_{x_{ijk}}(h))u_{x_{i,j,k}} \\ &+ (2\mu+\lambda)u_{x_{i+1,j,k}} + \mu u_{x_{i,j+1,k}} + \mu u_{x_{i,j,k+1}} \\ &+ \frac{\mu+\lambda}{4}(u_{y_{i+1,j+1,k}} - u_{y_{i+1,j-1,k}} - u_{y_{i-1,j+1,k}} + u_{y_{i-1,j-1,k}}) \\ &+ \frac{\mu+\lambda}{4}(u_{z_{i+1,j,k+1}} - u_{z_{i+1,j,k-1}} - u_{z_{i-1,j,k+1}} + u_{z_{i-1,j,k-1}}). \end{aligned}$$

Hierbei bezeichnet $C_{x_{ijk}}(h)$ den Wert, der sich bei der Auswertung der Ersatzlast in dem betrachteten Punkt (ih, jh, kh) ergibt. Er entspricht einer lokalen Federhärte und ist ein Maß dafür wie „fest“ ein Punkt fixiert ist. Das Penalty-Verfahren vereinfacht sich somit darauf, Strafparameter auf passenden Hauptdiagonalelemente der Systemmatrix zu addieren. Die Penalty-Methode ist somit sehr einfach und kosteneffizient umsetzbar. Nachteilig ist jedoch, dass a priori keine Aussagen über die benötigte Federhärte getroffen werden können und dass die Verschiebungen an fest eingespannten Punkten nicht komplett verschwinden, sondern nur auf ein vernachlässigbares Maß gedämpft werden.

2.3.2 Direkte Integration der Randbedingungen

Die zweite Hauptklasse der Immersed-Boundary-Methoden besteht aus den sogenannten Discrete-Forcing-Verfahren. Wie bei der Penalty-Methode ist auch hier die Idee, Einspannungen durch Manipulationen an der Systemmatrix und der rechten Seite in die Differentialgleichung zu integrieren. Im Gegensatz zu den Continuous-Forcing-Ansätzen verändert man hier aber erst nach der Diskretisierung das resultierende Gleichungssystem. Die einfachste Methode, die Randbedingung nach der Diskretisierung durchzusetzen, besteht darin, an allen Punkten des kartesischen Gitters, die in der eingespannten Menge liegen, die gegebenen Funktionswerte direkt vorzuschreiben. Dies ist beispielsweise möglich durch sogenannte semi-implizite Integration der Randbedingungen. Im Grunde basiert diese darauf, die Zeilen der Systemmatrix, die eingespannten Punkten entsprechen, durch Gleichungen der Form

$$u(x) = u_D(x)$$

zu ersetzen, was im Allgemeinen darin resultiert, Teile der Systemmatrix durch Einheitszeilen auszutauschen (analog zu Abschnitt 2.2.4). Dies macht das Verfahren zum einen äußerst verlässlich bezüglich der gewünschten Anpassung an komplexe Geometrien und zum anderen leicht umsetzbar.

2.4 Daten- und Löserstrukturen

Der folgende Abschnitt beruht auf [Wobker]. Wollen wir nun das lineare Gleichungssystem lösen, das aus der Diskretisierung der Lamé-Gleichung resultiert, so verwenden wir zum Beispiel das vorkonditionierte BiCGStab-Verfahren (vgl. [Göddeke]). Eine andere Möglichkeit, ein Lösungsverfahren zu wählen, ist das Richardson-Verfahren, da dieses ebenfalls keine Symmetrie der Systemmatrix voraussetzt. Die Geschwindigkeit und Effizienz dieser Verfahren lässt sich durch die Wahl des Vorkonditionierers beeinflussen. Bei der Wahl des Vorkonditionierers beschränken wir uns nicht auf die uns bereits bekannten elementaren Vorkonditionierer wie Jacobi und Gauß-Seidel, sondern überlegen, wie man diese Löser modifizieren kann, um die Matrixstruktur auszunutzen, die sich bei der Diskretisierung der Lamé-Gleichung und unter Verwendung verschiedener Nummerierungstechniken ergibt. Analog zum Modellproblem, das wir in der Vorlesung High Performance Computing und parallele Numerik I (vgl. [Göddeke]) untersucht haben, erhält die Systemmatrix bei zeilenweiser Nummerierung der Gitterpunkte auch hier Bandgestalt. Ein wesentlicher Unterschied besteht jedoch darin, dass wir nun keine skalare Differentialgleichung betrachten, sondern eine vektorwertige Differentialgleichung auf dem $\mathbb{R}^3$. Dies hat zur Folge, dass die Zahl der Unbekannten wächst. Statt wie bisher einer, haben wir nun drei Unbekannte pro Gitterpunkt. In jedem Gitterpunkt müssen wir die Verschiebungen in x-, y- und z-Richtung beachten. Hierdurch ergeben sich weitere Möglichkeiten zur Konstruktion effizienter Vorkonditionierer, wenn wir die Matrixstruktur genauer betrachten.

2.4.1 Ordnen nach Knoten (PDO)

Sei N die Anzahl der Gitterpunkte x_i in Ω, $i = 0, \ldots, N$. Eine erste Möglichkeit die Unbekannten anzuordnen nennt man Ordnen nach Knoten oder kurz PDO für *Pointwise Displacement Ordering*. Dazu setzt man

$$u = ((u_x(x_1), u_y(x_1), u_z(x_1)), \ldots, (u_x(x_N), u_y(x_N), u_z(x_N)))^\top .$$

So erhält man das lineare Gleichungssystem $K_{PDO} u = f$ mit

$$K_{PDO} = (\mathrm{K}_{rs})_{ij} = \mathrm{k}_{rs}(u_{i,j,k}), \quad r, s = 1, 2, 3, \quad i, j, k = 0, \ldots, N .$$

Hierbei ist $\mathrm{k}_{rs}(u_{i,j,k})$ wie in Gleichung (2.13) definiert und wir können die Matrix K schreiben als

$$K_{PDO} = \begin{pmatrix} K_{11} & \cdots & K_{1N} \\ \vdots & \ddots & \vdots \\ K_{N1} & \cdots & K_{NN} \end{pmatrix} \in \mathbb{R}^{3N \times 3N}$$

Sie besteht also aus $N \cdot N$ 3×3 Blöcken (s. Abb. 2.5).
Wir können beobachten, dass bei Benutzung der Methode der finiten Differenzen zweiter Ordnung höchstens sieben Blockmatrizen in einer Zeile ungleich der Nullmatrix sind [Halder]. Dies erscheint einleuchtend, wenn wir uns einmal genauer anschauen, was beim Ordnen der Unbekannten geschieht. Bildlich gesehen halten wir einen Punkt fest und betrachten dort nacheinander die Verschiebungen in x-, y- und z-Richtung. Dann erst gehen wir weiter zum nächsten Punkt und betrachten dort wieder die Verschiebungen in alle Richtungen und so weiter. Die Blockmatrizen in einer Zeile der Matrix entsprechen also den Kopplungen eines Knotens mit sich selbst und seinen sechs Nachbarn.

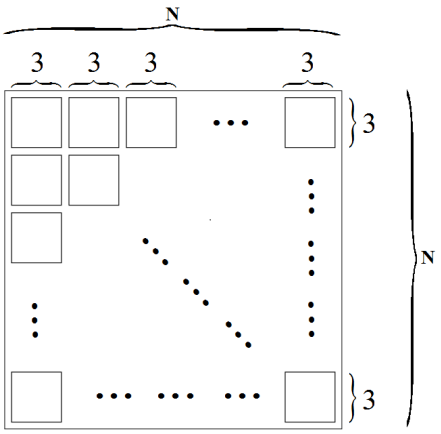


Abbildung 2.5: Blockstruktur der Matrix beim Ordnen nach Knoten

2.4.2 Ordnen nach Kopplung (SDO)

Eine andere Möglichkeit die Unbekannten anzuordnen nennt man Ordnen nach Kopplung oder kurz SDO für *Separate Displacement Ordering*. Dazu setzt man

$$u = ((u_x(x_1), \ldots, u_x(x_N)), (u_y(x_1), \ldots, u_y(x_N)), (u_z(x_1), \ldots, u_z(x_N)))^\top .$$

Das hierbei entstehende Gleichungssystem $K_{SDO}u = f$ wird nun definiert über

$$K_{SDO} = (\mathrm{K}_{rs})_{ijk} = \mathrm{k}_{rs}(u_{i,j,k}), \quad i,j,k = 0,\ldots,N, \quad r,s = 1,2,3\ ,$$

wobei wir wieder die $\mathrm{k}_{rs}(u_{i,j,k})$ aus Gleichung (2.13) verwenden.
Im Gegensatz zu PDO halten wir bei SDO erst eine Verschiebungsrichtung fest und betrachten dann die Punkte. Das heißt, wir betrachten erst in jedem Punkt nur die Verschiebung in x-Richtung. Anschließend gehen wir die Gitterpunkte noch einmal komplett durch, betrachten nun aber die Verschiebungen in y-Richtung. Für die z-Richtung verfahren wir analog. So beschreibt jeder Matrixblock K_{rs}, welchen Einfluss die Verschiebung in Richtung i auf die in Richtung j hat. Die Matrix K_{SDO} erhält die Blockstruktur aus Abbildung 2.6:

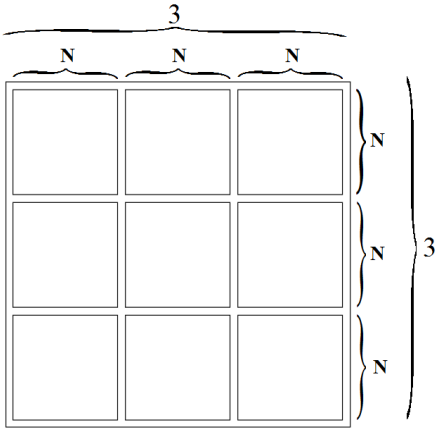


Abbildung 2.6: Blockstruktur der Matrix beim Ordnen nach Kopplung

Wir beschreiben die Matrix im Folgenden durch

$$K_{SDO} = \begin{pmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{pmatrix} \in \mathbb{R}^{3N\times 3N}\ .$$

2.4.3 Löserstrukturen

Ordnet man für die Diskretisierung der Lamé-Gleichung die Unbekannten punktweise, so ist für die Wahl des Vorkonditionierers die sich ergebene Blockstruktur der Matrix nicht ausschlaggebend. Man verwendet also bekannte Vorkonditionierer wie Jacobi oder Gauß-Seidel. Eine andere Herangehensweise wäre die Wahl eines ILU-artigen Vorkonditionierers, der den Vorteil hat, dass er ohne jegliche Vorkenntnis über die Struktur der Matrix zuverlässige Ergebnisse liefert.
Nehmen wir also an, dass unsere Variablen nach Kopplung sortiert sind. Nun können wir die bekannten Vorkonditionierer intuitiv auf die Blockgestalt der Matrix K übertragen. Als Löser verwenden wir wieder das Richardson- oder das vorkonditionierten BiCGStab-Verfahren. Hierbei ist die Neuerung, dass nun so genannte Block-Vorkonditionierer eingesetzt werden.

Block-Jacobi

Anstelle der Diagonalen der Matrix wählen wir nun also die Blockmatrizen, die auf der Diagonalen stehen, als Vorkonditionierungsmatrix

$$K_{\text{BJac}} := \begin{pmatrix} K_{11} & 0 & 0 \\ 0 & K_{22} & 0 \\ 0 & 0 & K_{33} \end{pmatrix}.$$

Aus der Sicht dieses Vorkonditionierers handelt es sich bei der Matrix K also um eine 3×3-Matrix. Zur Anwendung eines Jacobi-Schrittes benötigen wir nun die Inversen der Diagonal-Blockmatrizen. Dazu müssen wir in jedem Schritt drei lineare Gleichungssysteme lösen:

$$K_{11}u_x = f_x, \quad K_{22}u_y = f_y, \quad K_{33}u_z = f_z$$

Dies geschieht mit Hilfe der bekannten Löser. Wir haben nun also ein System aus Vorkonditionierern. Dabei bezeichnen wir den Block-Vorkonditionierer auch als äußeren und die Vorkonditionierer für die einzelnen Matrixblöcke als innere Vorkonditionierer.
Da der Block-Jacobi-Vorkonditionierer nur eine 3×3-Matrix „sieht“ ist die Konditionszahl der äußeren Vorkonditionierungsmatrix von der Feinheit des Gitters h unabhängig. Für weitere Einflüsse der beiden Nummerierungstechniken verweisen wir auf [Axelsson]. Man beachte aber, dass dort die Methode der finiten Elemente zur Diskretisierung verwendet wurde.
Da die entstandenen Systeme unabhängig voneinander sind, können sie parallel gelöst werden. Dies verfolgen wir jedoch nicht weiter, da wir mit dem Einsatz von GPU und Multicore-CPU (vgl. Kapitel 2.5) eine höhere Parallelität erreichen können.

Block-Gauß-Seidel

Analog zu unserem Vorgehen bei dem Jacobi-Vorkonditionierer erweitern wir nun den Gauß-Seidel-Vorkonditionierer auf einen solchen in Blockgestalt:

$$K_{\text{BGS}} := \begin{pmatrix} K_{11} & 0 & 0 \\ K_{21} & K_{22} & 0 \\ K_{31} & K_{32} & K_{33} \end{pmatrix}$$

Dies führt auf die zu lösenden Gleichungssysteme:

$$K_{11}u_x = f_x \quad K_{22}u_y = f_y - K_{21}u_x \quad K_{33}u_z = f_z - K_{31}u_x - K_{32}u_y$$

Zur Lösung dieser Gleichungssysteme verwenden wir auch hier wieder die uns bekannten Löser.
Verglichen mit einem Block-Jacobi-Vorkonditionierer haben wir mehr Rechenaufwand pro Iterationsschritt, da durch die Berücksichtigung der Matrixblöcke K_{21}, K_{31} und K_{32} in unserem Vorkonditionierer drei Matrix-Vektor-Multiplikationen mehr durchzuführen sind. Außerdem entfällt die Parallelität der

inneren Probleme.
Analog zu dem normalen Gauß-Seidel-Vorkonditionierer kann man bei Verwendung von K_{BGS} anstelle von K_{BJac} bessere Konvergenz beobachten.
Verwendet man einen Block-Gauß-Seidel-Vorkonditionierer innerhalb von BiCGStab, muss man auf die Wahl der Startlösung achten, da eine schlechte Wahl dazu führen kann, dass das Verfahren abbricht, ohne gelöst zu haben. Dieses Problem wird in [Wobker] näher erläutert.

2.5 Multicore-CPUs und GPUs

Betrachtet man die Lamé-Gleichung im dreidimensionalen Fall, stellt man schnell fest, dass das diskretisierte Problem mit kleiner werdender Schrittweite h schnell größer wird. Dies hat zur Folge, dass auch die Rechenzeit, die zur Lösung des Systems benötigt wird, stark anwächst, da die Matrizen größer werden und die Konvergenzrate der betrachteten BiCGStab- und Richardson-Löser von der Schrittweite h abhängt. Hier stößt man schnell an die Grenzen einer Rechnerstruktur mit einer einfachen CPU.
Um die Laufzeit der Löser zu optimieren, lernten wir im Laufe der Vorlesung High Performance Computing und Parallele Numerik (vgl. [Göddeke]) Multicore-CPUs kennen. Die Idee hierbei ist, dass man Operationen parallel auf verschiedenen Kernen ausführen lässt. Betrachtet man beispielsweise das Bilden eines Skalarprodukts zweier Vektoren, so würde dies bedeuten, dass jeder Kern einen gewissen „Abschnitt“ der beiden Vektoren erhält und mit diesen „kleinen“ Vektoren ein Skalarprodukt ausrechnet. Anschließend wird das Skalarprodukt als Summe der Ergebnisse der einzelnen Kerne berechnet.
Während des Studienprojekts lernten wir eine weitere Rechnerstruktur, die GPU, kennen. Hier liegt die selbe Idee wie bei Multicore-CPUs zugrunde. Ein wesentlicher Unterschied besteht jedoch in der Anzahl der sogenannten *Threads*, die parallel rechnen können. GPUs sind aufgrund ihrer Rechnerarchitektur (vgl. Abb.2.7) besonders auf die Optimierung des Durchsatzes vieler gleicher Aufgaben ausgelegt. Sie sind gegenüber Multicore-CPUs also besonders dann im Vorteil, wenn Rechenoperationen die Datenparallelität ausnutzen. Will man Rechnungen auf häufig wechselnden Daten ausführen, ist eine Multicore-CPU die geschicktere Wahl, da diese für die Optimierung der Latenz ausgelegt ist. Die Hauptmotivation, die Berechnungen von der CPU auf die GPU zu verlagern, war also die Möglichkeit, durch massive Parallelität bessere Laufzeiten zu erzielen, was für das Hochleistungsrechnen sehr interessant ist. Dabei haben wir mit CUDA auf einer *GeForce GTX 560 Ti*-Grafikkarte und mit OpenMP auf dem für uns bereitgestellten *Intel-Core i7 970*-Prozessor gerechnet. Somit konnten wir auf der CPU mit 6 Kernen rechnen und es standen uns auf der GPU 8 Multiprozessoren mit je 48 CUDA Cores zur Verfügung – wir konnten dort also mit bis zu 12288 Threads parallel rechnen. Für detailliertere Informationen über GPU-Architekturen und das CUDA-Programmiermodell verweisen wir auf [Garland].

Abbildung 2.7: Architektur einer Multicore-CPU und einer GPU [NVIDIA]

Kapitel 3

Projektorganisation und -planung

In diesem Kapitel beschreiben wir zunächst unsere Überlegungen hinsichtlich der zum effizienten Lösen der vorgegebenen Problemstellung notwendigen Rahmenbedingungen. Diese Vorüberlegungen bezogen sich auf die verwendete Hardware, die Assemblierung der Systemmatrix und die Wahl der Löser und Vorkonditionierer. Des Weiteren wurden erste Ideen zu möglichen Vergleichen entwickelt und diskutiert. Außerdem wird in diesem Abschnitt die Umsetzung der in Kapitel 2 eingeführten Theorie vorgestellt. Um die einzelnen Teilgebiete voneinander unabhängig zu gestalten und die nötigen Einstellungen von außen vornehmen zu können, wurde unserem Programmcode ein modulares Konzept zu Grunde gelegt, was in den einzelnen Bereichen entsprechend realisiert wurde.

3.1 Entwicklungsphasen

Um ein so komplexes Projekt umzusetzen, bedarf es einiger Zeit der Vorbereitung und Einarbeitung hinsichtlich der benötigten Kenntnisse. Diese mussten vor allem in den Bereichen Programmieren und effizientes numerisches Lösen erworben und vertieft werden.
Vorbereitend hörten wir zunächst die Vorlesung High Performance Computing und Parallele Numerik, die uns ein grundlegendes und weiterführendes Verständnis der Programmiersprache C, insbesondere die Benutzung von OpenMP, sowie verschiedener Vorkonditionierungs- und Lösungstechniken für dünn besetzte lineare Gleichungssysteme vermittelt hat.
Als Startschuss für die eigentliche Projektarbeit diente eine Seminarphase zu Beginn des zweiten Semesters unseres Studienprojekts, in der problembezogene theoretische Grundlagen gelegt wurden (vgl. Kapitel 2). Diese Grundlagen bezogen sich jedoch nicht nur auf die oben behandelten Aspekte, sondern auch auf den Umgang mit einem SVN-Server zur Ermöglichung effizienter Gruppenarbeit, der Visualisierung der betrachteten Testfälle mittels Paraview sowie der Erarbeitung der GPU-Programmierung mithilfe der CUDA-Bibliothek.
Hierauf aufbauend folgte die Implementierungsphase, während der wir uns zunächst in die Kleingruppen Assemblierung, Vorkonditionierung und GPU-Programmierung aufteilten. Neben der CUDA- und CUBLAS-Bibliothek, als Hilfsmittel für die GPU-Programmierung, verwendeten wir auch Teile der MKL-Bibliothek, um einen stärkeren Vorkonditionierer einsetzen zu können. Zum Kompilieren haben wir die GNU Compiler Collection genutzt. Die Koordination der Teilgruppen erfolgte in wöchentlichen Treffen mit unserem Projektbetreuer. Mit dem Voranschreiten der Implementierung änderte sich die Gruppenaufteilung kontinuierlich, um neue Aufgaben, wie die Visualisierung in Paraview [1], anzugehen.
Nach der erfolgreichen Implementierung konnte die Test- und Berichtphase beginnen, wozu wir uns in zwei gleichgroße Gruppen aufteilten.

[1] `http://www.paraview.org/`

3.2 Modularisierte Implementierung

3.2.1 Speicherverwaltung und numerische lineare Algebra

Das Ziel der Implementierung war, die Nutzung unserer eigenen Bibliotheken so universell wie möglich zu gestalten. Einen wesentlichen Bestandteil des Codes bilden dabei Verfahren und Operationen der numerischen linearen Algebra. Als erstes wurde eine Unterscheidung zwischen Speicherallokierung (`memory.c`) und Rechenoperationen (`numlinalg.c`) sowohl für konventionelle als auch für Blockmatrizen vorgenommen. In dem Modul `memory.c` kann der Nutzer von außen eingeben, ob Speicher auf der CPU oder GPU allokiert wird. So gibt es beispielsweise eine Funktion `mallocVec`, die optional den Speicherplatz eines Vektors auf der CPU oder GPU allokiert. Dazu wird ein Parameter `target` eingegeben, der entweder den Wert `TARGET_CPU` oder `TARGET_GPU` annehmen kann. Eine „Verteilerfunktion“ verweist dann auf die entsprechende `_CPU`- oder `_GPU`-Variante der Routine. In dem Modul `numlinalg.c` gibt es ähnliche Funktionen, die anhand der Datenallokation der übergebenen Vektoren entscheiden, ob eine Operation, zum Beispiel das Berechnen von Skalarprodukten oder Normen, auf der CPU oder GPU ausgeführt werden soll. Die eigentliche Implementierung der Operationen basiert dann auf CUDA und CUBLAS für GPU beziehungsweise OpenMP für Multicore-CPUs.
Der Unterschied von normaler zu Block-Implementierung in den Modulen `numlinalg.c` und `memory.c` besteht hierbei aus einer zusätzlichen Schleife über die Blöcke. Die Arbeitsweise der Module `numlinalg-block.c` und `memory-block.c` besteht darin, für jeden Matrixblock die entsprechende konventionelle Routine aufzurufen und wenn nötig anschließend zusammenzuführen. Soll beispielsweise ein Skalarprodukt zweier Block-Vektoren berechnet werden, so werden erst drei Skalarprodukte gebildet. Jeder Block in einem Vektor wird hierbei als normaler Vektor an die Skalarprodukt-Routine des Moduls `numlinalg.c` weitergegeben. Dort werden die konventionellen Skalarprodukte berechnet und anschließend aufsummiert.
Es können alle bestehenden Funktionen in dem Modul `numlinalg.c` direkt genutzt werden, ohne bei der Benutzung der Routinen darauf achten zu müssen, ob der Speicher auf der GPU oder CPU allokiert ist. Somit reduziert sich die Umstellung von CPU auf GPU nur noch auf das Setzen eines Schalters beim Kompilieren (`GPU=YES / NO`). Fundamental ist dabei das von uns verwendete *Makefile*. Dort besteht nicht nur die Möglichkeit, eine Entscheidung zwischen CPU und GPU zu treffen, sondern auch auf einer Multicore-CPU zu rechnen. Dies geschieht über eine entsprechende Eingabe in der Kommandozeile (`OPENMP=YES / NO`). Außerdem kann mittels eines weiteren Schalters zwischen zwei verschiedenen Kompiliermodi gewählt werden. `OPT=NO` stellt einen Debugging-Modus zur Verfügung, indem verschiedene Abfragen überprüfen, ob zum Beispiel Speicher für die eingegebenen Daten reserviert wurde oder ob zwei zu addierende Vektoren die gleiche Größe haben. Das führt dazu, dass das Programm unter Umständen abgebrochen wird beziehungsweise nur dann auszuführen ist, wenn die Basisoperationen korrekt oder mit korrekten Parametern aufgerufen werden. Der optimierte Modus `OPT=YES` verwendet spezifische Compilereinstellungen für die zugrunde liegende Hardware. Im Debugmodus werden zusätzlich Hilfsprogramme wie `valgrind` unterstützt, die die Fehlersuche vereinfachen.

3.2.2 Assemblierung

Die Frage, ob wir Matlab oder C als Programmiersprache verwenden wollten, war sehr schnell geklärt. Wir erstellten anfänglich einen Code in Matlab, der dazu diente, das Konzept der Matrixassemblierung zu validieren und Vergleichslösungen zu erstellen, um die komplexere Umsetzung in C zu überprüfen. Für den Testfall einer festen Einspannung aller Seitenflächen des Einheitswürfels wurde eine zusätzliche Matrix assembliert. Hier konnte nur die Gitterweite h von außen verändert werden. Diese Matrix diente dem Zweck, die strukturbezogenen Vorkonditionierer und Löser schon vorzeitig testen zu können. Die eigentliche Assemblierungsroutine findet sich in der Datei `lameassembly.c` und wurde sehr flexibel gestaltet. Sie ermöglicht es sowohl verschiedene Testfälle zu simulieren, das heißt unterschiedliche Einspannungs- und Belastungssituationen, als auch mit den verschiedenen Immersed-Boundary-

Methoden (vgl. Kapitel 2.3) und Nummerierungstechniken (vgl. Kapitel 2.4) zu experimentieren.

3.2.3 Löser und Vorkonditionierer

Die in Abschnitt 3.2 vorgestellte Struktur wird auch bei den Lösern und Vorkonditionierern ausgenutzt, da dort auf die jeweiligen Operationen auf der CPU oder GPU verwiesen wird. Sobald man ein `TARGET` gewählt hat, übernehmen die Verteilerfunktionen das Aufrufen von CPU- oder GPU-Routinen und dem Anwender wird diese Aufgabe abgenommen. Es reicht also aus, nur einen Löser zu implementieren, da dieser dann sowohl auf der CPU als auch auf der GPU ausgeführt werden kann.
Bei der Assemblierung der Löser und Vorkonditionierer müssen wir trotzdem eine Unterscheidung treffen. Diese besteht in den zwei verschiedenen Möglichkeiten, die Unbekannten anzuordnen (vgl. Kapitel 2.4), was zu unterschiedlichen Matrixstrukturen führt. Aus diesem Grund wurden zwei „Pakete“ mit entsprechenden Lösern und Vorkonditionieren realisiert.

PDO-Nummerierung

Ordnen wir die Unbekannten nach Knoten, so ist die Blockgestalt der entstehenden Systemmatrix nicht ausschlaggebend. Hier können wir also die Löser und Vorkonditionierer nutzen, die schon im Rahmen der Vorlesung High Performance Computing und Parallele Numerik (vgl. [Göddeke]) vorgestellt und implementiert wurden, insbesondere das Richardson- und das vorkonditionierte BiCGStab-Verfahren mit einem Jacobi- oder Gauß-Seidel-Vorkonditionierer. Hierbei kann das Jacobi-Verfahren auch parallel gerechnet werden. Zusätzlich wurde der ILU(0)-Vorkonditionierer aus der MKL-Bibliothek eingebunden. Die Löser wurden in der Datei `solver.c` und die Vorkonditionierer in `precon.c` realisiert.

SDO-Nummerierung

Um die Blockstruktur der Systemmatrix im SDO-Format auszunutzen, haben wir die oben genannten Löser auf das Blockformat erweitert. Dies wurde realisiert, indem Matrix-Vektor-Operationen durch das Einfügen einer zusätzlichen Schleife über die Blöcke angepasst wurden. Außerdem wurden die in Abschnitt 2.4.3 beschriebenen Vorkonditionierer implementiert. Als innerer Löser wird eine leicht modifizierte Version der Löser für das PDO-Format verwendet. So steht auch hier die oben genannte Auswahl an inneren Vorkonditionierern zur Verfügung. Analog zur PDO-Nummerierung wurden auch hier die äußeren Löser in der Datei `solver-block.c` und die Vorkonditionierer in `precon-block.c` implementiert.

Auch bei den Lösern und Vorkonditionieren kann vorab gewählt werden, welche Kombinationen man benutzen will. Zuerst wird der Löser über den Parameter `solver` festgelegt, der die Werte `0` für das vorkonditionierte BiCGStab-Verfahren oder `1` für das Richardson-Verfahren annehmen kann. Die dort benötigten Vorkonditionierer werden über einen weiteren Parameter gewählt. Hier stehen der Jacobi-, der Gauß-Seidel- und der ILU(0)-Vorkonditionierer zur Verfügung. Wählt man den Jacobi-Vorkonditionierer, besteht zusätzlich die Möglichkeit auf der Multicore-CPU oder auf der GPU parallel zu rechnen.
Bei Block-Lösern muss auch der innere Löser und Vorkonditionierer gewählt werden. Außerdem müssen hier Parameter, wie die Anzahl maximaler Iterationen, für den inneren Löser übergeben werden. Diese werden in einer Struktur `param` zusammengeführt. So ist es möglich von außen alle nötigen Einstellungen vorzunehmen und die zugehörigen Parameter festzulegen.

3.2.4 Applikationen

Die oben beschriebenen Codefragmente wurden schließlich in zwei Applikationen zusammengeführt. Um diese Teile einfach in die Programme einzubinden, haben wir eine „Header“-Datei `lamesolver.h` erstellt, in der die benötigten Dateien verlinkt wurden. So ist es möglich, nur diese eine Datei einbinden

zu müssen, um auf alle Routinen und somit die gesamte Funktionalität unserer Simulationsumgebung zugreifen zu können. Die Parameter der unterschiedlichen Funktionen können in den beiden Applikationenen festgelegt werden. An dieser Stelle können also verschiedene Testfälle (Belastungen und Geometrien), Löser und Vorkonditionierer ausgewählt werden. Die erste Applikation ist die `DIAapp`, die das aus der diskretisierten Lamé-Gleichung entstandene lineare Gleichungssystem löst. Hierbei wird das PDO-Format verwendet. Im Gegensatz dazu löst die `BLOCKapp` dieses Problem unter Verwendung der SDO-Nummerierung.

3.3 Testkategorien

Mit fortschreitender Implementierung haben wir diskutiert und festgelegt, welche Vergleiche und Tests wir uns als Schwerpunkte setzen. Diese liegen in den oben behandelten Themen.
Im Bereich Hardware haben wir uns für den Vergleich von (Multicore-)CPUs und GPUs entschieden, um herauszufinden, ab wann das Rechnen auf der GPU deutlich effizienter ist als auf der CPU. Besonders interessant ist hierbei, wie groß der Speedup von der GPU zur CPU ist.
In der Kategorie Assemblierung vergleichen wir die Varianten SDO und PDO, die in Kapitel 2.4 vorgestellt wurden. Im Vordergrund steht hier vor allem der Vergleich zwischen „normalen“ und Blocklösern. Dabei geht es hauptsächlich um das Gegenüberstellen von Iterationszahlen, Konvergenzraten und Exaktheit der Lösungen. Also wird untersucht, in wie weit das Lösen innerhalb der Matrixblöcke den Löser insgesamt beeinflusst.
Zuletzt werden verschiedene Vorkonditionierer untersucht, um deren Qualität und Beitrag zum Lösen des Gleichungssystems zu bewerten.

Kapitel 4

Numerische Tests

Im Fokus dieses Kapitels stehen die numerischen Tests, die wir mit unserer Simulationssoftware durchgeführt haben. Anhand dieser Versuche werden wir zum einen demonstrieren, dass unser Code ordnungsgemäß arbeitet, und zum anderen auf die Frage eingehen, inweit sich die theoretischen Vorhersagen aus Kapitel 2 mit unseren Ergebnissen decken. In den Abschnitten 4.3 und 4.4 werden wir des Weiteren untersuchen, durch welche Strategien sich die Berechnungen am besten beschleunigen lassen und wie sinnvoll die Block-Vorkonditionierungs- und Block-Lösungsansätze in der Praxis sind.

4.1 Validierung des Programmcodes

In dem folgenden Abschnitt werden wir anhand zweier Testfälle belegen, dass unser Programm sowohl mathematisch als auch physikalisch sinnvolle Lösungen berechnet. Das erste Szenario besteht aus einer einfachen Belastungssituation, in der sich die Verschiebungen analytisch bestimmen lassen. Mit Hilfe dieser Referenzlösung werden wir untersuchen, wie sich der Diskretisierungsfehler bei kleiner werdender Schrittweite verhält und wie gut die von uns berechneten Näherungen die exakte Lösung approximieren. Darüber hinaus werden wir an dieser Stelle auf die Frage eingehen, inwiefern sich die theoretischen Voraussagen bezüglich dieser Aspekte anhand unserer Ergebnisse bestätigen lassen. Im zweiten Testfall werden wir eine einfache Zug- und Druckprobe simulieren und demonstrieren, dass sich die von uns berechneten Verschiebungen in guter Näherung mit der Theorie des Hookeschen Gesetzes decken.

4.1.1 Verhalten gegenüber einer analytischen Referenzlösung

Um eine analytische Referenzlösung unter vertretbarem Aufwand berechnen zu können, werden wir uns im Folgenden auf die Situation beschränken, dass die gesamte Oberfläche des belasteten Blockes fest gelagert ist, also alle Punkte in der Oberfläche homogenen Dirichlet-Randbedingungen unterliegen. Die zu lösende Differentialgleichung vereinfacht sich damit zu

$$\begin{aligned} -(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) &= f \quad \text{in }]0,1[^3 \\ u &= 0 \quad \text{auf } \partial]0,1[^3 \end{aligned}$$

Geben wir uns nun eine Verschiebungsfunktion u vor, die an den Rändern des Würfels verschwindet und in einem Bereich liegt, in dem lineares Materialverhalten vorausgesetzt werden kann, beispielsweise

$$u(x,y,z) = 10^{-9} \begin{pmatrix} x(x-1)y(y-1)z(z-1) \\ x(x-1)y(y-1)z(z-1) \\ x(x-1)y(y-1)z(z-1) \end{pmatrix}$$

so lässt sich durch Einsetzen in die Differentialgleichung die rechte Seite f bestimmen, die zu genau dieser Lösung passt, und man erhält einen Spezialfall der Lamé-Gleichung, zu dem eine analytische

Lösung bekannt ist. Für das obige u ergibt sich etwa

$$f(x,y,z) =$$
$$-10^{-9}\,(\mu+\lambda)\begin{pmatrix} 2y(y-1)z(z-1)+(2x-1)(2y-1)z(z-1)+(2x-1)y(y-1)(2z-1) \\ (2x-1)(2y-1)z(z-1)+2x(x-1)z(z-1)+x(x-1)(2y-1)(2z-1) \\ (2x-1)y(y-1)(2z-1)+x(x-1)(2y-1)(2z-1)+2x(x-1)y(y-1) \end{pmatrix}$$
$$-10^{-9}\mu\begin{pmatrix} 2y(y-1)z(z-1)+2x(x-1)z(z-1)+2x(x-1)y(y-1) \\ 2y(y-1)z(z-1)+2x(x-1)z(z-1)+2x(x-1)y(y-1) \\ 2y(y-1)z(z-1)+2x(x-1)z(z-1)+2x(x-1)y(y-1) \end{pmatrix}$$

Im Folgenden werden wir genau diese Belastungsfunktion f voraussetzen. Für das Material Stahl, das heißt für die Elastizitätskonstanten (vgl. Tabelle 2.1)

$$E = 2.1\cdot 10^{11}\text{ Pa}, \qquad \nu = 0.29, \qquad \mu = \frac{E}{2(1+\nu)}, \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)},$$

entspricht die Referenzlösung u dann einer Verschiebung entlang der Würfeldiagonalen, wie sie in Abbildung 4.1 zu sehen ist. Die Abbildungen 4.2 a) bis d) zeigen nun die mit unserer Simulationssoftware bestimmten Lösungen zu den Schrittweiten $h = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ und $\frac{1}{16}$. Rein optisch ist gut zu erkennen, dass unsere Näherungen für kleiner werdende Schrittweiten die exakte Verschiebungsfunktion immer besser approximieren. Bei der Diskretisierung wurde hier die in Kapitel 2.4.1 vorgestellte PDO-Gitternummerierung benutzt, gelöst wurde durch ein BiCGStab-Iterationsverfahren mit Gauß-Seidel-Vorkonditionierung (Skalierungsparameter $\omega = 1$, Abbruchkriterium $\varepsilon = 10^{-12}$, Startlösung $x_0 = (10^{-9})_{i=1,\dots,N}$, mit OpenMP und sechs Kernen). Die Färbungen sind ein Maß für die euklidische Norm des Verschiebungsvektors u in den einzelnen Gitterpunkten.

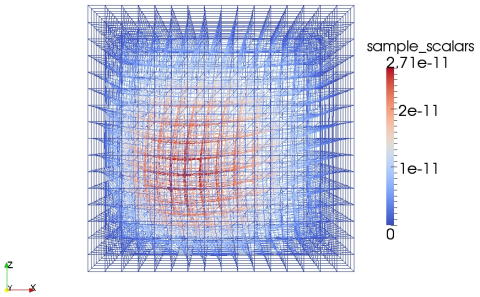
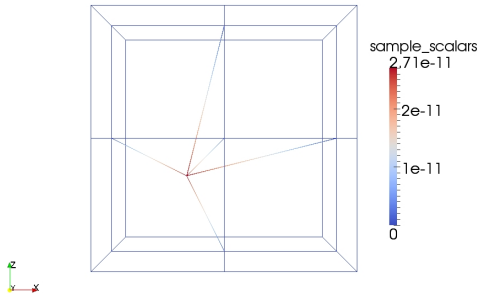
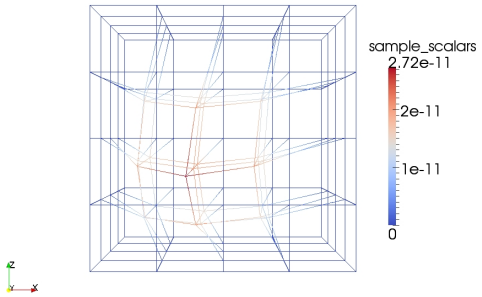


Abbildung 4.1: Analytische Referenzlösung ausgewertet auf einem Gitter der Schrittweite $h = \frac{1}{16}$

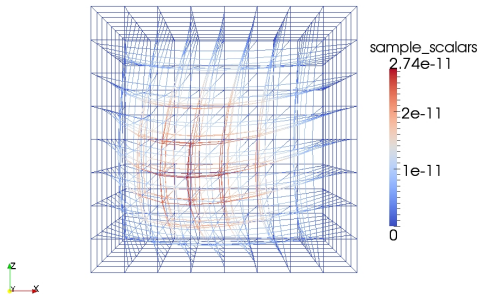
Mathematisch zeigt sich die oben beschriebene Qualitätsverbesserung dadurch, dass mit der Schrittweite h auch der relative Konsistenzfehler $e = \frac{\|Au_h - f\|_2}{\|u_h\|_2}$ abnimmt. Da bei der Diskretisierung der Differentialgleichung (vgl. Kapitel 2.2) ausschließlich Differenzenquotienten zweiter Ordnung verwendet wurden, ist hier zu erwarten, dass auch e asymptotisch wie $const \cdot h^2$ gegen Null geht. Wie in Abbildung 4.3 a) zu sehen ist, lässt sich dieses Verhalten auch in der Praxis beobachten. Es ist deutlich zu erkennen, dass - zumindest in guter Näherung und für kleine h - bei einer Halbierung der Schrittweite der Diskretisierungsfehler e um den Faktor 4 reduziert wird. Abbildung 4.3 b) zeigt darüber hinaus die Entwicklung des relativen Fehlers zwischen der von uns berechneten Näherung und der exakten Lösung (bezüglich der Euklidischen Norm). Offensichtlich pendelt sich dieser für kleine Schrittweiten bei tolerablen sechs Prozent ein. Zugrundegelegt wurde für diese beiden Tests erneut die oben beschriebene Testkonfiguration zu verschiedenen Schrittweiten. Die kontinuierlichen Graphen entstanden durch lineare Interpolation.



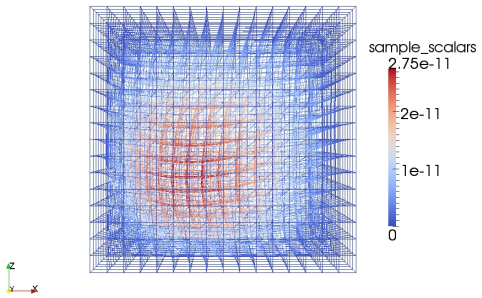
(a) Näherungslösung zu $h = \frac{1}{2}$



(b) Näherungslösung zu $h = \frac{1}{4}$

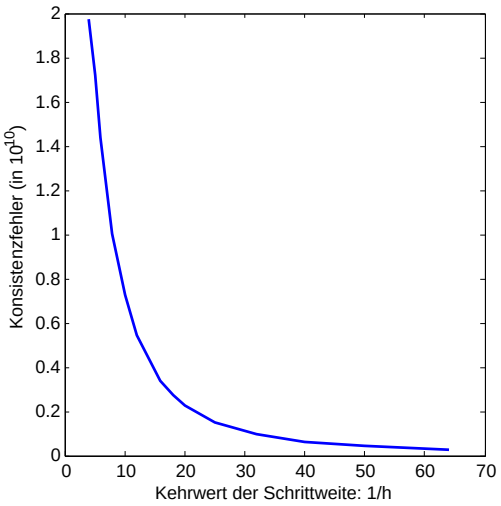


(c) Näherungslösung zu $h = \frac{1}{8}$

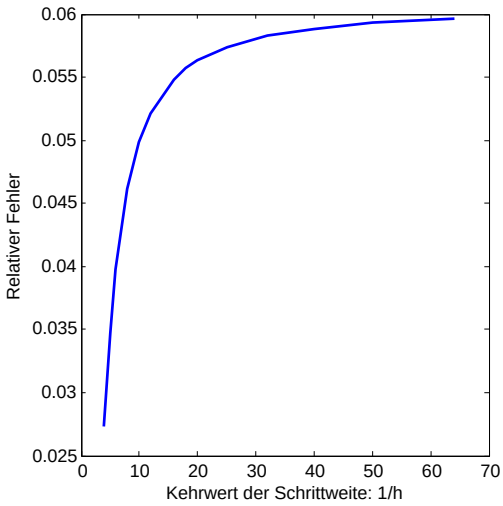


(d) Näherungslösung zu $h = \frac{1}{16}$

Abbildung 4.2: Berechnete Näherungen zu verschiedenen Schrittweiten (Testkonfiguration siehe oben). Um die Deformationen sichtbar zu machen, wurden die Verschiebungen mit dem Faktor 10^{10} skaliert.



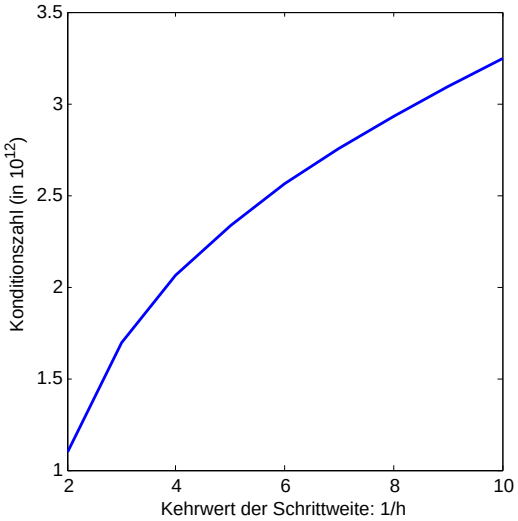
(a) Asymptotisches Verhalten des Konsistenzfehlers



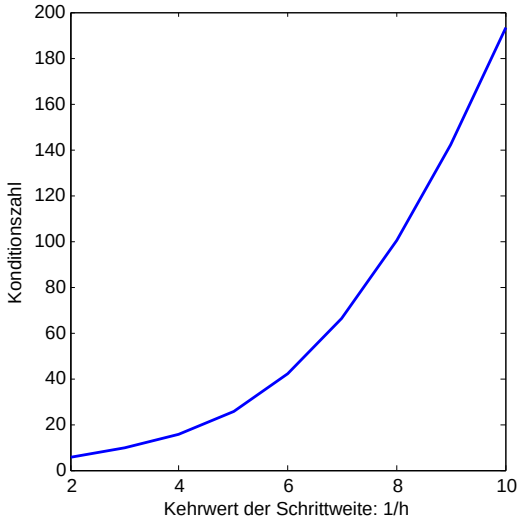
(b) Entwicklung des relativen Fehlers zur exakten Referenzlösung (bezüglich der Euklidischen Norm)

Abbildung 4.3: Fehlerentwicklungen

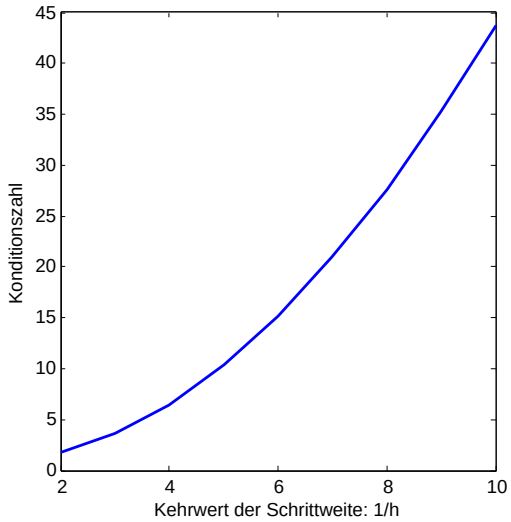
Neben dem Verhalten des Konsistenzfehlers genügt auch die Entwicklung der Konditionszahl unserer Systemmatrix den theoretischen Voraussagen. Da es sich bei der Lamé-Gleichung um eine Differentialgleichung zweiter Ordnung handelt und wir eine konsistente Diskretisierung mittels Differenzenquotienten zweiter Ordnung gewählt haben, ist hier generell zu erwarten, dass die Kondition mit abnehmender Schrittweite wie $const \cdot \frac{1}{h^2}$ gegen Unendlich strebt (vgl. Kapitel 2.2). Bei unserer Matrix ist aber zusätzlich zu beachten, dass durch die semi-implizite Integration der Dirichlet-Randbedingungen Einheitszeilen eingeschoben werden, die dieses Verhalten stören. Aus diesem Grund lässt sich die vorhergesagte Entwicklung $\text{cond}(A) = \mathcal{O}(h^{-2})$ lediglich für Materialien mit moderaten Eigenschaften beobachten, während sich für Extremfälle ein anderes Verhalten einstellt beziehungsweise das vorhergesagte Wachstum erst bei sehr kleinen Schrittweiten zur Geltung kommt und somit für die Praxis nicht relevant ist. Man betrachte hierzu Abbildung 4.4.



(a) Stahl: $E = 2.1 \cdot 10^{11}$, $\nu = 0.29$



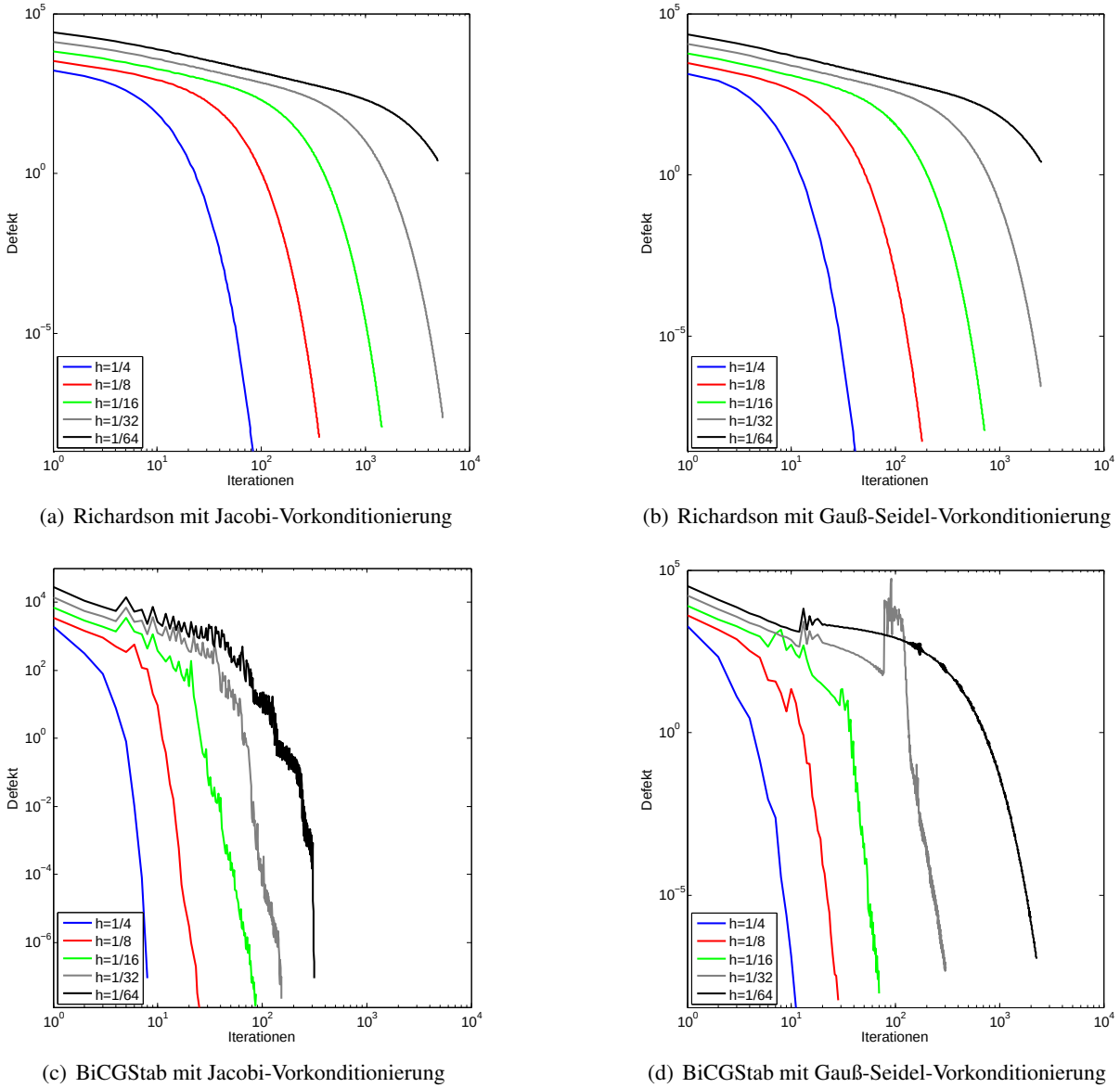
(b) Gummi: $E = 0.1$, $\nu = 0.49$



(c) Moderater Referenzwerkstoff: $E = \nu = 0.29$

Abbildung 4.4: Verhalten der Konditionszahl für kleiner werdende Schrittweiten in der oben beschriebenen Belastungssituation unter Verwendung verschiedener Materialien. Man beachte die grundsätzlich unterschiedlichen Entwicklungen für die Extremfälle Stahl ($E \gg 1$) und Gummi ($\nu \approx 0.5$), sowie die quadratische Entwicklung für einen Werkstoff mit moderaten Elastizitätseigenschaften.

Das Wachstum der Konditionszahlen beeinflusst dabei direkt das Konvergenzverhalten der iterativen Löser, man vergleiche hierzu die Abbildungen 4.5 a), b), c) und d). Sowohl beim Richardson- als auch beim BiCGStab-Verfahren nehmen hier die Iterationszahlen mit abnehmender Schrittweite zu, bei ersterem proportional zu $\frac{1}{h^2}$, bei zweiterem unregelmäßig. Darüber hinaus zeigt sich deutlich, dass der Jacobi- die Gauß-Seidel-Vorkonditionierung und dem Richardson-Verfahren der BiCGStab-Löser vorzuziehen ist. Des Weiteren sind im Konvergenzverlauf des BiCGStab-Lösers deutlich die typischen Instabilitäten zu erkennen, die sich durch die Maschinengenauigkeit und die rundungsfehleranfällige A-Orthogonalitätsbedingung ergeben (Details hierzu finden sich etwa bei [Göddeke]). Das Richardson-Verfahren zeigt hingegen eine kontinuierliche Defektreduktion, die jedoch auf Kosten der Geschwindigkeit geht. Die theoretischen Voraussagen lassen sich anhand unserer Ergebnisse also bestätigen.



(a) Richardson mit Jacobi-Vorkonditionierung

(b) Richardson mit Gauß-Seidel-Vorkonditionierung

(c) BiCGStab mit Jacobi-Vorkonditionierung

(d) BiCGStab mit Gauß-Seidel-Vorkonditionierung

Abbildung 4.5: Defektreduktion für verschiedene Löser-Vorkonditionierer-Kombinationen. Berechnet wurde das obige Modellproblem zu den Schrittweiten $h = \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}$ und $\frac{1}{64}$ unter Benutzung der PDO-Gitternummerierung. Man beachte das Zusammenbrechen des aufgebauten Krylov- Unterraumes bei der Verwendung des BiCGStab- mit dem Gauß-Seidel-Verfahren bei $h = \frac{1}{32}$. Restarts erfolgten hier bei schlecht konditionierten Divisionen, das heißt im Fall „Divisor $<$ Divident $\cdot\, 10^{-14}$“.

Zusammenfassend können wir an dieser Stelle zu dem Ergebnis kommen, dass sich alle Resultate, die wir in dem oben beschriebenen Szenario unter Benutzung unseres Codes bestimmt haben, so verhalten, wie es die Theorie vorhersagt und die analytische Referenzlösung erfordert. Sowohl die Assemblierungsroutinen als auch die iterativen Löser arbeiten mathematisch korrekt und lassen sich somit im Folgenden guten Gewissens auch auf kompliziertere Problemstellungen anwenden.

4.1.2 Vergleich mit dem Hookeschen Gesetz

Um zu demonstrieren, dass die von uns berechneten Näherungen nicht nur der mathematischen Theorie genügen, sondern auch den Gesetzen der zugrundeliegenden Physik, werden wir in dem folgenden Abschnitt eine einfache Zug- und Druckprobe simulieren und die Ergebnisse mit den Voraussagen des Hookeschen Gesetzes vergleichen. Wir betrachten dazu einen an der Unterseite fest eingespannten Block, der an seiner Oberseite durch eine Normalspannung belastet wird. Setzen wir voraus, dass keine Volumenkräfte auftreten, so ergibt sich in dieser Situation als zu lösende Differentialgleichung

$$-(\mu+\lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = 0 \quad \text{in }]0,1[^3$$

$$u_z = 0 \quad \text{auf } \Gamma_D :=]0,1[^2 \times \{0\}$$

$$\sigma \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ t \end{pmatrix} \quad \text{auf } \Gamma_{N1} :=]0,1[^2 \times \{1\}$$

$$\sigma n = 0 \quad \text{auf } \Gamma_{N2} := \partial]0,1[^3 \setminus (\Gamma_D \cup \Gamma_{N1})$$

wobei mit σ der in Kapitel 2.1 eingeführte Spannungstensor, mit n der lokale Normalenvektor und mit t die eingeprägte Spannung bezeichnet wird. Im Gegensatz zu der Problemstellung aus Abschnitt 4.1.1 liegen hier also nicht nur homogene Dirichlet-Randbedingungen, sondern auch inhomogene und homogene Neumann-Randbedingungen an den belasteten beziehungsweise freien Würfelflächen Γ_{N1} beziehungsweise Γ_{N2} vor. Da es sich bei diesem Szenario um eine eindimensionale Belastungssituation handelt, bietet es sich an, zur Berechnung der Verschiebungen ein linear elastisches Materialverhalten entlang der z-Koordinate vorauszusetzen und die Deformationen in die anderen Raumrichtungen zu vernachlässigen. Diese Annahmen führen auf das Hookesche Gesetz, das sich für das obige Problem auf den folgenden Ausdruck reduziert (vgl. [Kessel]):

$$u_z(x,y,z) = \frac{t \cdot z}{E}$$

Die Verschiebungen der Gitterpunkte hängen in diesem vereinfachten Modell also linear von ihrer Höhe ab und sind unabhängig von der Position in der jeweiligen x-y-Ebene. Wählen wir als Material Gummi mit den Elastizitätskonstanten

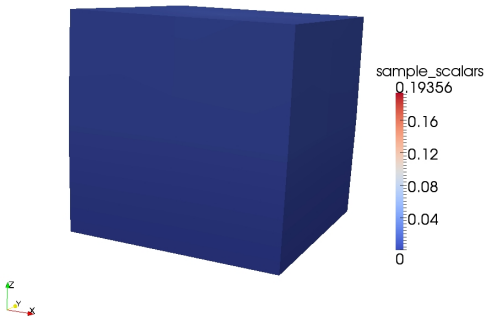
$$E = 0.1 \text{ Pa}, \qquad \nu = 0.49,$$

$$\mu = \frac{E}{2(1+\nu)} = 0.03 \text{ Pa}, \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} = 1.64 \text{ Pa}$$

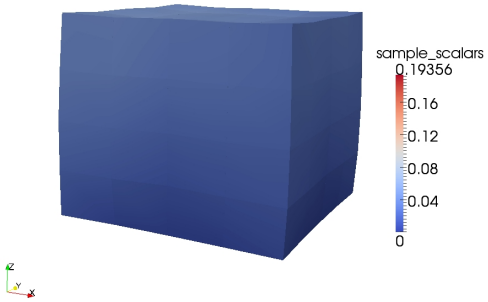
und als Belastungsspannung $t = -2 \cdot 10^{-2}$, so erhalten wir die konkrete Verschiebungsfunktion

$$u_z(x,y,z) = -0.2 \cdot z$$

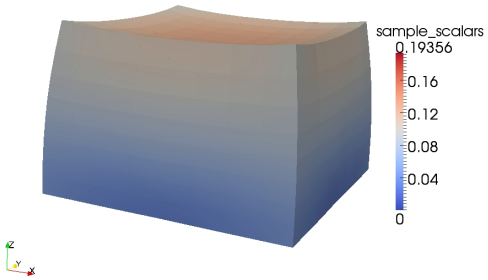
In Abbildung 4.6 sind die Ergebnisse unserer Berechnungen in dieser Belastungssituation zu sehen. Benutzt wurde hier ein BiCGStab-Löser mit Gauß-Seidel-Vorkonditionierung (Skalierungsparameter $\omega = 1$, Abbruchkriterium $\varepsilon = 10^{-8}$, Startlösung $x_0 = (10^{-1})_{i=1,\dots,N}$, mit OpenMP und sechs Kernen) und die aus Kapitel 2.4.1 bekannte PDO-Gitternummerierung. Die Färbungen geben an, wie stark sich die Gitterpunkte an der jeweiligen Stelle verschoben haben, wobei blau keine Verschiebung und rot die maximale Verschiebung codiert.



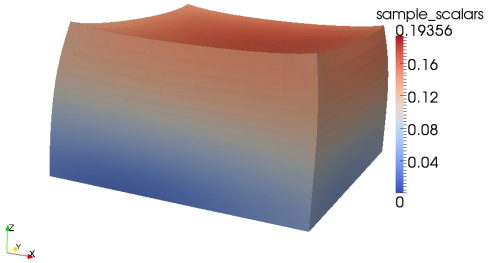
(a) Näherungslösung zu $h = \frac{1}{2}$



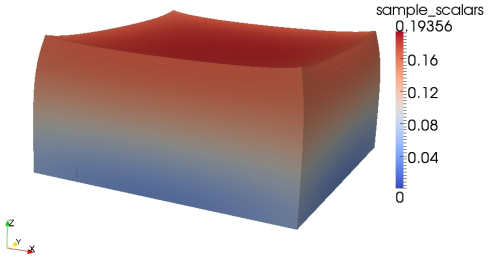
(b) Näherungslösung zu $h = \frac{1}{4}$



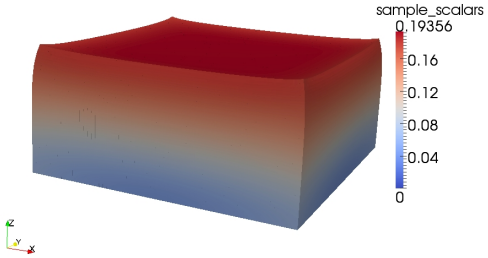
(c) Näherungslösung zu $h = \frac{1}{8}$



(d) Näherungslösung zu $h = \frac{1}{16}$



(e) Näherungslösung zu $h = \frac{1}{32}$



(f) Näherungslösung zu $h = \frac{1}{64}$

Abbildung 4.6: Simulation des Druckversuches mit verschiedenen Schrittweiten h (genaue Testkonfiguration und Belastungsart siehe oben). Zur besseren Visualisierung wurden die Verschiebungen in die drei Raumrichtungen mit dem Faktor 2.5 skaliert.

Da die Lamé-Gleichung im Gegensatz zum Hookeschen Gesetz auch Verschiebungen entlang der x- und der y- Koordinate miteinbezieht, treten hier Deformationen in alle Raumrichtungen auf. Um unsere Lösungen dennoch „fair“ mit den Voraussagen des eindimensionalen Modells vergleichen zu können, werden wir im Folgenden ausschließlich die mittlere Sehne des Blockes betrachten, das heißt die Menge $S := \{(0.5, 0.5, z) \in [0,1]^3 |\ z \in [0,1]\}$. Diese weist naturgemäß die geringsten Querverschiebungen auf und kommt deshalb den Voraussagen des Hookeschen Gesetzes am nächsten. Wie Abbildung 4.7

zeigt, ergeben sich an diesen Gitterpunkten genau die erwarteten Resultate. Zum einen verhält sich unsere Näherung bei kleiner werdenden Schrittweiten auf der Menge S zunehmend linear, zum anderen konvergieren die Verschiebungen auf der Sehne für $h \to 0$ gegen die Werte, die das Hookesche Gesetz vorhersagt. Unsere Lösung zeigt somit sowohl quantitativ als auch qualitativ das Verhalten, das die physikalische Theorie von ihr erwartet. Wir können also davon ausgehen, dass unser Programm auch für kompliziertere Geometrien und Belastungssituationen Resultate produziert, die den Ergebnissen realer Experimente nahekommen.

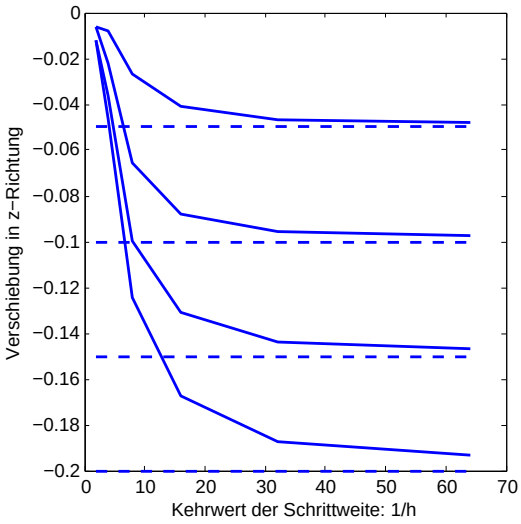


Abbildung 4.7: Verschiebungen entlang der z-Achse in den Gitterpunkten $(0.5, 0.5, 0.25)$, $(0.5, 0.5, 0.5)$, $(0.5, 0.5, 0.75)$ und $(0.5, 0.5, 1)$ (von oben nach unten) zu verschiedenen Schrittweiten h. Gestrichelt sind die Werte gekennzeichnet, die das Hookesche Gesetz an diesen Stellen vorhersagt.

4.2 Test des Penalty-Verfahrens

Im Folgenden wollen wir untersuchen, wie sich die beiden Immersed-Boundary-Methoden, die wir zur Integration von Dirichlet-Randbedingungen im Falle komplizierter Geometrien programmiert haben, zueinander verhalten (vgl. Kapitel 2.3). Als Testfall benutzen wir hier einen Zug- und Druckversuch analog zu Kapitel 4.1.2. Die zugrundeliegende Differentialgleichung lautet damit

$$-(\mu + \lambda)\,\mathrm{grad}(\mathrm{div}(u)) - \mu\,\mathrm{div}(\mathrm{grad}(u)) = 0 \quad \text{in }]0,1[^3$$

$$u = 0 \quad \text{auf } \Gamma_D :=]0,1[^2 \times \{0\}$$

$$\sigma \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ t \end{pmatrix} \quad \text{auf } \Gamma_{N1} :=]0,1[^2 \times \{1\}$$

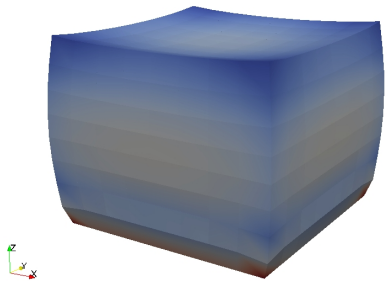
$$\sigma n = 0 \quad \text{auf } \Gamma_{N2} := \partial]0,1[^3 \setminus (\Gamma_D \cup \Gamma_{N1})$$

mit σ als dem Spannungstensor aus Kapitel 2.1, n als dem lokalen Normalenvektor und t als der auf der Oberseite des Blockes eingeprägten Spannung. Als Material setzen wir erneut Gummi voraus, das heißt die Elastizitätskonstanten

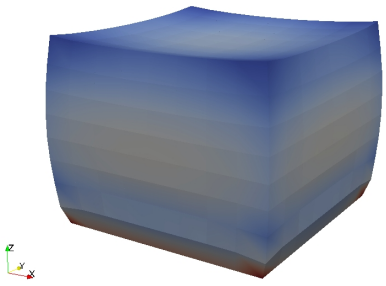
$$E = 0.1 \text{ Pa}, \qquad \nu = 0.49,$$

$$\mu = \frac{E}{2(1+\nu)} = 0.03 \text{ Pa}, \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} = 1.64 \text{ Pa}$$

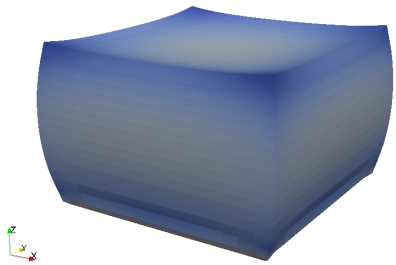
Belasten werden wir mit $t = -2 \cdot 10^{-2}$. Zum Vergleich des Penalty-Verfahrens mit der semi-impliziten Integration sind in Abbildung 4.8 nun die Ergebnisse einiger Berechnungen zu sehen, die wir zur Simulation dieses Szenarios durchgeführt haben.



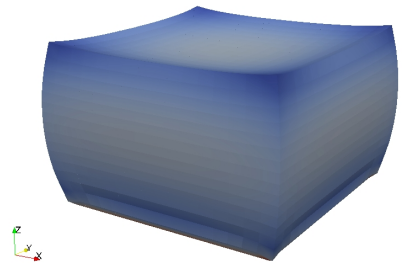
(a) Lösung bei semi-impliziter Integration zu $h = \frac{1}{8}$



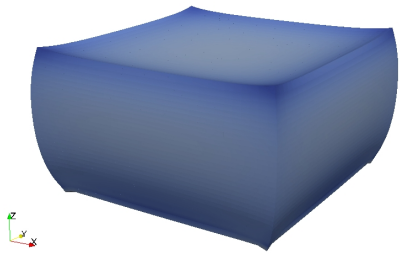
(b) Lösung der Penalty-Methode zu $h = \frac{1}{8}$



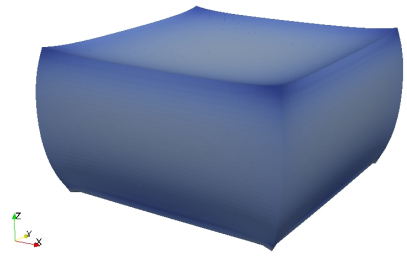
(c) Lösung bei semi-impliziter Integration zu $h = \frac{1}{16}$



(d) Lösung der Penalty-Methode zu $h = \frac{1}{16}$



(e) Lösung bei semi-impliziter Integration zu $h = \frac{1}{32}$



(f) Lösung der Penalty-Methode zu $h = \frac{1}{32}$

Abbildung 4.8: Simulation des Druckversuches mit verschiedenen Schrittweiten h und unter Verwendung der beiden unterschiedlichen Immersed-Boundary-Methoden. Zur besseren Visualisierung wurden die Verschiebungen mit dem Faktor 2.5 skaliert. Die Färbungen sind hier ein Maß für die internen Spannungen.

Zur Lösung wurde in beiden Fällen der BiCGStab-Löser mit Gauß-Seidel-Vorkonditionierung verwendet ($\omega = 1$, Abbruchkriterium $\varepsilon = 10^{-8}$, Startlösung $x_0 = (10^{-1})_{i=1,\dots,N}$, PDO-Nummerierung, mit OpenMP und sechs Kernen). Der Strafparameter betrug $C = 10^{12}$, als Distributionsfunktion konnte hier aufgrund der gitterkonformen Einspannung die Kronecker-Funktion δ verwendet werden. Rein optisch sind die beim Penalty-Verfahren zwangsläufig auftretenden Verschiebungen an der eigentlich fixierten unteren Würfelfläche nicht zu erkennen, sowohl qualitativ als auch quantitativ zeigen die Lösungen der beiden Verfahren in guter Näherung dasselbe Verhalten. Wie Abbildung 4.9 zeigt, lässt sich diese intuitive Einschätzung auch mathematisch belegen. Es ist hier deutlich zu erkennen, dass der relative Fehler zwischen den Näherungen gegenüber den anderen Fehlerquellen (vgl. Kapitel 4.1) vernachlässigbar klein ist.

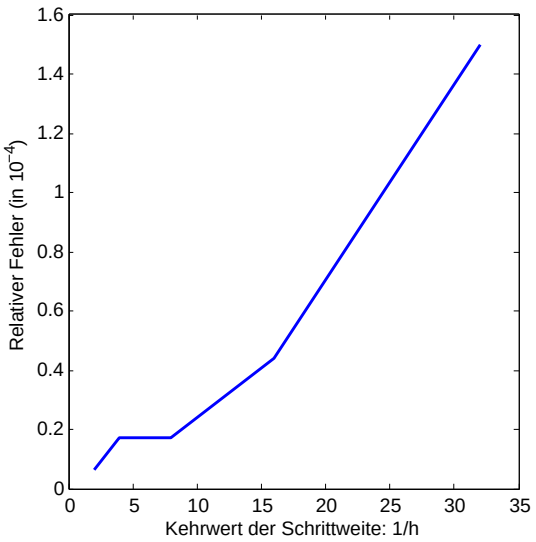
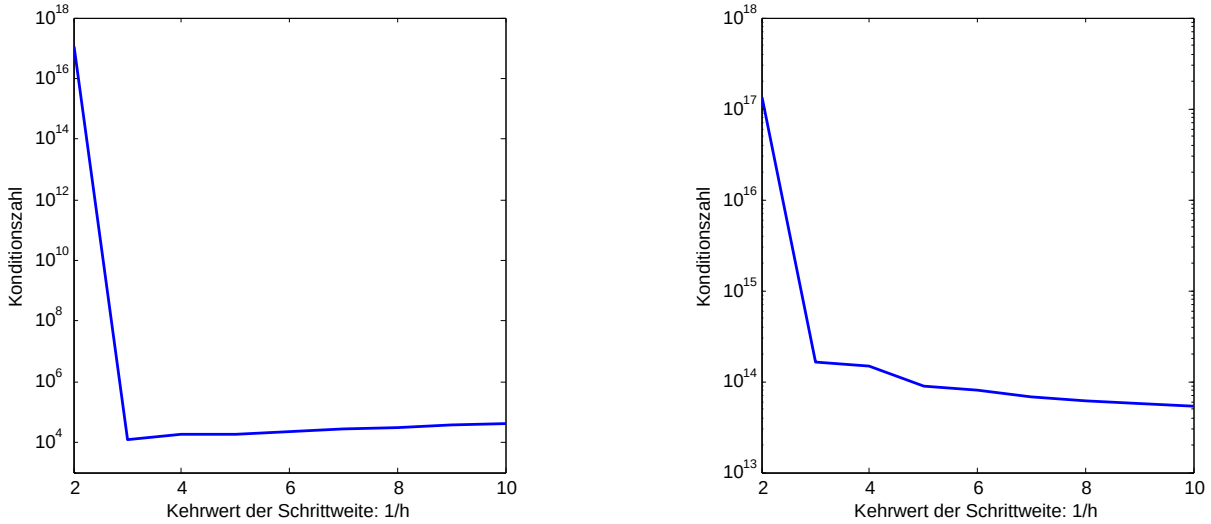


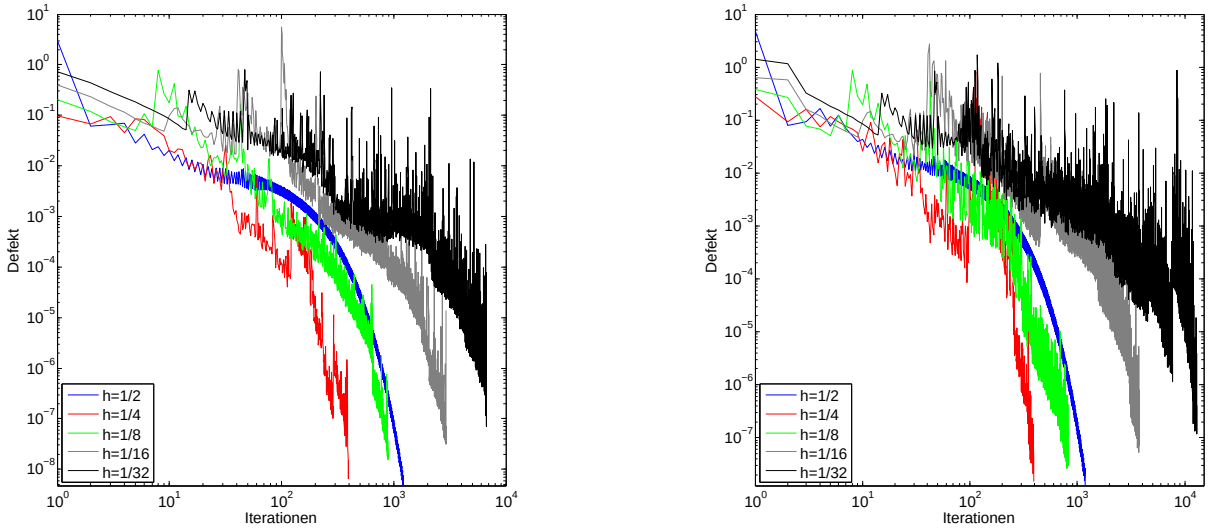
Abbildung 4.9: Verhalten des relativen Fehlers $e = \frac{\left\| u_{\text{Penalty}} - u_{\text{Semi-implizit}} \right\|_2}{\left\| u_{\text{Semi-implizit}} \right\|_2}$ zwischen den Näherungslösungen des Penalty-Verfahrens und der semi-impliziten Integrationsmethode der Randbedingungen.

Die enorme Höhe des Strafparameters $C = 10^{12}$ ermöglicht es zu garantieren, dass die ungewollten Verschiebungen an der unteren Würfelseite im Gleitkommarauschen untergehen. Sie verursacht jedoch auch dieselben Komplikationen, die sich bei Materialien mit extremen Elastizitätseigenschaften ($E \gg 1$ oder $\nu \approx 0.5$) einstellen (vgl. Kapitel 4.1.1). So ist beispielsweise die Konditionszahl der Systemmatrix in dieser Konfiguration des Penalty-Verfahrens gegenüber der bei der semi-impliziten Integration deutlich höher, wie in Abbildung 4.10 klar zu erkennen ist. Für kleine Schrittweiten zieht dies wiederum ein schlechteres Konvergenzverhalten der iterativen Löser nach sich (s. Abbildung 4.11). Um die Konditionszahl auf ein vertretbares Maß zu reduzieren, kann der Strafparameter C systematisch verringert werden. Zu beachten ist hierbei aber, dass mit der Höhe von C zwangsläufig auch die Genauigkeit an den Einspannungsstellen sinkt. Ein vernünftiger Kompromiss lässt sich hier zumeist nur heuristisch oder experimentell bestimmen. Es zeigt sich an dieser Stelle deutlich, dass sich die Penalty-Methode nur unter hohem Aufwand auf unsere dreidimensionalen Probleme anwenden lässt. Die Stabilisierung des Verfahrens erfordert zahlreiche Vorversuche und zieht im Falle komplizierterer Geometrien aufwendige Berechnungen zur optimalen Verteilung der Strafparameter nach sich. Die semi-implizite Integrationsmethode ist im direkten Vergleich unkomplizierter und effektiver. Aus diesem Grund werden wir uns in den folgenden Testfällen auch auf diese beschränken.



(a) Semi-implizite Integration (b) Penalty Verfahren

Abbildung 4.10: Entwicklung der Konditionszahl der Systemmatrix bei Verwendung der beiden Integrationsmethoden. Man beachte die äußerst schlechte Kondition bei $h = \frac{1}{2}$, die durch die entartete Kopplungssituation bei Verwendung der einseitigen Differenzenquotienten auf einem $3 \times 3 \times 3$-Gitter entsteht (siehe Kapitel 2.2).

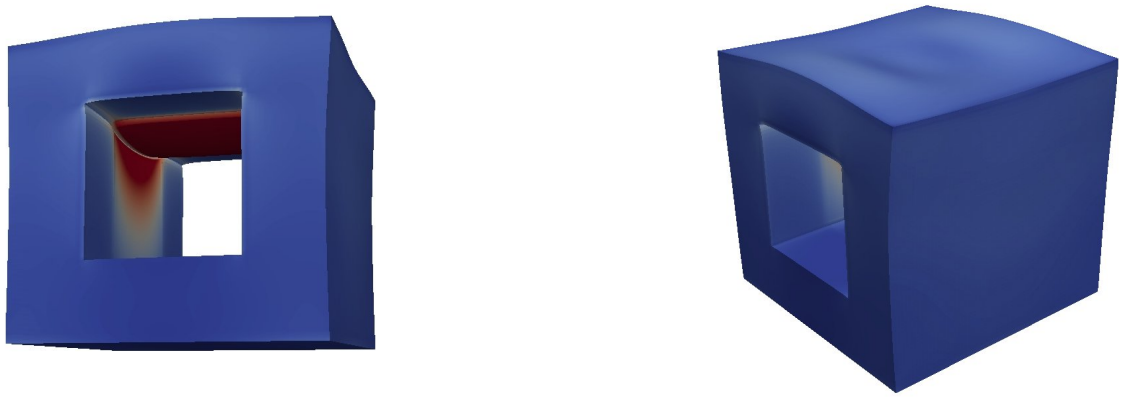


(a) Semi-implizite Integration (b) Penalty Verfahren

Abbildung 4.11: Konvergenzverhalten bei Verwendung der beiden Integrationsmethoden. Simuliert wurde die oben beschriebene Problemsituation unter Verwendung der Schrittweiten $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ und $\frac{1}{32}$. Es ist deutlich zu erkennen, dass sich bei der Verwendung des Penalty-Verfahrens mehr Instabilitäten ergeben als bei der semi-impliziten Integrationsmethode. Restarts erfolgten hier bei schlecht konditionierten Divisionen, das heißt in dem Fall „Divisor $<$ Divident $\cdot 10^{-14}$ “.

4.3 Geschwindigkeitsmessungen

In den folgenden drei Abschnitten werden die Laufzeiten, die zum Lösen der Lamé-Gleichung auf der GPU, der Multicore-CPU sowie der Singlecore-CPU benötigt werden, gegenübergestellt. Um alle Zeitmessungen sinnvoll miteinander vergleichen zu können, haben wir uns auf den Testfall eines Metallblocks, aus dem mittig ein Loch ausgeschnitten wurde, beschränkt. Dieser Block wird von oben in negative z-Richtung durch die Spannung $t = 1000\,\text{Pa}$ belastet und ist am Boden in alle Richtungen fest eingespannt, wie in Abbildung 4.12 zu sehen ist. Als Material haben wir dabei Stahl gewählt, also gilt für das Elastizitätsmodul $E = 2.1 \cdot 10^{11}$ und die Querkontraktionszahl $\nu = 0.29$. Als Löser haben wir das BiCGStab-Verfahren mit Jacobi-Vorkonditionierung verwendet und mit 512 Threads pro Threadblock auf der GPU, beziehungsweise mit 6 CPU-Threads mit OpenMP gerechnet. Getestet wurden die Schrittweiten $h \in \left\{\frac{1}{12}, \frac{1}{16}, \frac{1}{20}, \ldots, \frac{1}{64}\right\}$.



(a) Ansicht von schräg unten

(b) Ansicht von schräg oben

Abbildung 4.12: Testfall bei Schrittweite $h = \frac{1}{64}$

4.3.1 Singlecore- und Multicore-CPUs

Wir vergleichen als erstes die Rechenzeiten der Multicore-CPU mit der Singlecore-CPU (im Folgenden kurz M-CPU und S-CPU), um herauszufinden, inwiefern es sich lohnt, M-CPUs in der Hochleistungsrechnung zu verwenden. Das zu lösende Problem ist in der Einleitung zu Kapitel 4.3 beschrieben. Die Vermutung liegt nahe, dass mit der sechsfachen Anzahl an Kernen ein Speedup von fünf bis sechs erzielt werden kann, was sich in der Praxis aber nicht beobachten lässt. An der Speedupkurve in Grafik 4.13 erkennt man, dass für größere Probleme lediglich ein Speedup von ungefähr drei erreicht wird. (vgl. Tabelle 4.1). Das liegt zum Einen daran, dass die Verwendung von OpenMP immer mit einem gewissen Overhead beispielsweise zur Synchronisierung verbunden ist, der erst für große Datenmengen amortisiert wird und sich somit negativ auf die Laufzeiten auswirkt. Dieser Effekt tritt beispielsweise in Rechenoperationen wie der Norm oder dem Skalarprodukt auf, bei denen mehrere Kerne auf gleichen Daten arbeiten, sich also gegenseitig synchronisieren müssen und mehr Operationen als in einer seriellen Implementierung durchgeführt werden. In unserem Programmcode ist dies mit einer `reduction` gelöst, welche die Schreibrechte der Kerne organisiert. Zum Anderen stellt die Speicherbandbreite eine weitere physikalische Einschränkung dar, die, unabhängig davon, ob in einer Routine Speicherkonflikte auftreten, die Rechenzeiten beeinflusst. Wie sich diese bei einer bestimmten Operation auswirkt, ist abhängig von der sogenannten arithmetischen Intensität, die definiert ist als Rechenoperationen pro Speicherzugriff. Je kleiner die arithmetische Intensität ist, desto größer müsste die Speicherbandbreite sein, um die theoretisch maximale Rechenleistung einer Architektur auszunutzen. Anhand des Beispiels der

Vektoraddition $z = x + y$ lässt sich der Einfluss der arithmetischen Intensität anschaulich erläutern: Addieren wir zwei Vektoren der Länge N, so müssen N reine Rechenoperationen durchgeführt werden, bei denen zusätzlich jeweils zwei Lese- und eine Schreiboperation anfallen. Insgesamt kommt man also auf $3N$ Speicheroperationen. Die arithmetische Intensität ergibt sich hieraus zu $\frac{1N}{3N} = \frac{1}{3}$. Betrachten wir nun den von uns verwendeten Rechner (Westmere-CPU, DDR3-Speicher, s. Kapitel 2.5), so hat dieser eine theoretische Maximalleistung von 78.6 GFLOP/s bei einer Speicherbandbreite von 25.6 GB/s. Bei Rechnung in doppelter Genauigkeit impliziert die arithmetische Intensität von $\frac{1}{3}$ also, dass selbst bei optimaler Ausnutzung der Speicherbandbreite nur $\frac{25.6}{8} \cdot \frac{1}{3} \approx 1.1$ GFLOP/s erreicht werden können, was circa 1.4% der maximalen Rechenleistung entspricht. Andersherum formuliert bräuchten wir also $(\frac{1}{3})^{-1} \cdot 8 \cdot 7.68 \cdot 10^{10} \approx 1.8$ TB/s Bandbreite. Uns steht aber nur eine Bandbreite von 25.6 GB/s zur Verfügung, sodass unsere Berechnungen gebremst werden. Dieses prinzipielle Problem nennt man auch *„memory wall“*. Für nähere Informationen verweisen wir auf [Dongarra]. Nichtsdestotrotz empfiehlt sich der Einsatz von M-CPUs, da sich selbst ein Speedup um den Faktor drei bei den hohen Rechenzeiten der S-CPUs auszahlt.

h	S-CPU	M-CPU	Speedup
$\frac{1}{12}$	24.63	4.24	5.81
$\frac{1}{16}$	80.87	17.45	4.634
$\frac{1}{20}$	452.66	124.16	3.646
$\frac{1}{24}$	1001.08	299.86	3.338
$\frac{1}{28}$	2020.88	589.81	3.426
$\frac{1}{32}$	3625.52	1067.8	3.395
$\frac{1}{36}$	6183.89	1841.75	3.358
$\frac{1}{40}$	9768.01	2956.26	3.304
$\frac{1}{44}$	15346.47	4615.83	3.324
$\frac{1}{48}$	22504.36	6882.08	3.27
$\frac{1}{52}$	31557.42	9968.42	3.166
$\frac{1}{56}$	45222.37	14139.32	3.198
$\frac{1}{60}$	63720.62	19988.19	3.188
$\frac{1}{64}$	94946.13	29876.81	3.18

Tabelle 4.1: Laufzeitvergleich in Sekunden von S-CPU und M-CPU

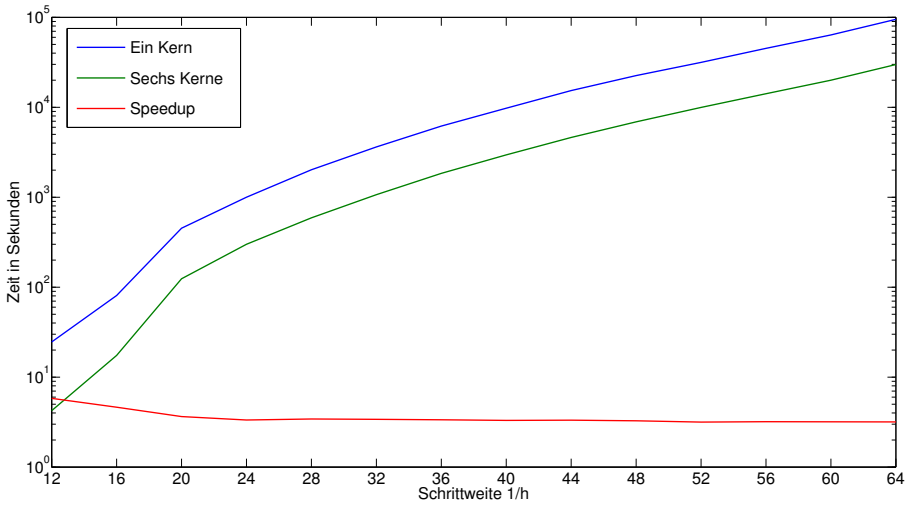


Abbildung 4.13: Zeitmessungen der seriellen Laufzeit gegen OpenMP Parallelisierung

4.3.2 Multicore-CPU und GPU

Nachdem wir im vorherigen Kapitel herausgearbeitet haben, inwiefern die Benutzung von OpenMP bereits die Rechenzeit verringert, wird im Folgenden betrachtet, welchen weiteren Speedup wir durch die Nutzung der GPU erreichen können. Um sicher zu stellen, dass die GPU-Implementierung die gleichen Ergebnisse wie die CPU-Implementierung liefert, wurde diese zunächst durch den Vergleich von Iterationszahlen und Defektverläufen überprüft. Anschließend wurde das in Kapitel 4.3 beschriebene Problem gelöst. Hierbei wurde die PDO-Nummerierung zur Assemblierung der Matrix und eine variable Schrittweite h verwendet. In Abbildung 4.14 sind die Ergebnisse der Zeitmessungen sowie der Speedup zu sehen.

h	CPU	GPU	Speedup
$\frac{1}{12}$	4.24	6.7	0.632
$\frac{1}{16}$	17.45	11.92	1.464
$\frac{1}{20}$	124.16	21.01	5.908
$\frac{1}{24}$	299.86	37.22	8.056
$\frac{1}{28}$	589.81	63.74	9.254
$\frac{1}{32}$	1067.8	105.64	10.108
$\frac{1}{36}$	1841.75	166.75	11.045
$\frac{1}{40}$	2956.26	265.7	11.126
$\frac{1}{44}$	4615.83	401.46	11.498
$\frac{1}{48}$	6882.08	589.11	11.682
$\frac{1}{52}$	9968.42	843.12	11.823
$\frac{1}{56}$	14139.32	1179.22	11.99
$\frac{1}{60}$	19988.19	1608.48	12.427
$\frac{1}{64}$	29876.81	2186.62	13.663

Tabelle 4.2: Laufzeitvergleich in Sekunden von CPU und GPU

Auffällig ist, dass die CPU nur für sehr grobe Schrittweiten h schneller ist als die GPU, für $h = \frac{1}{12}$ ist sie ungefähr doppelt so schnell. Dies lässt sich dadurch erklären, dass die Dimension der Systemmatrix noch recht klein ist und somit nicht so viele Rechnungen ausgeführt werden. Deshalb macht die zusätzliche Zeit, die benötigt wird, um die Daten von der CPU auf die GPU zu kopieren einen größeren Teil der Gesammtlaufzeit aus. Verkleinert man die Schrittweite h nur geringfügig, ist die GPU bereits schneller. Schon bei $h = \frac{1}{16}$ ist die CPU langsamer und der erreichte Speedup wächst bei weiterer Schrittweitenverkleinerung rasch auf elf an. Danach steigt der Speedup langsamer, aber streng monoton bis auf etwa 13 für die Schrittweite $h = \frac{1}{64}$. Dies lässt sich gut am Speedupverlauf ablesen (s. Abb. 4.14).
In Kapitel 4.3.1 haben wir festgestellt, dass der Einsatz von M-CPUs uns bei unseren Berechnungen einen Speedup von etwa drei bringt. Weiterhin wurde nun beobachtet, dass die Verwendung von GPUs eine weitere Verbesserung der Laufzeit um den Faktor 13 bewirkt. Insgesamt bedeutet dies einen Speedup von 40 gegenüber der S-CPU.

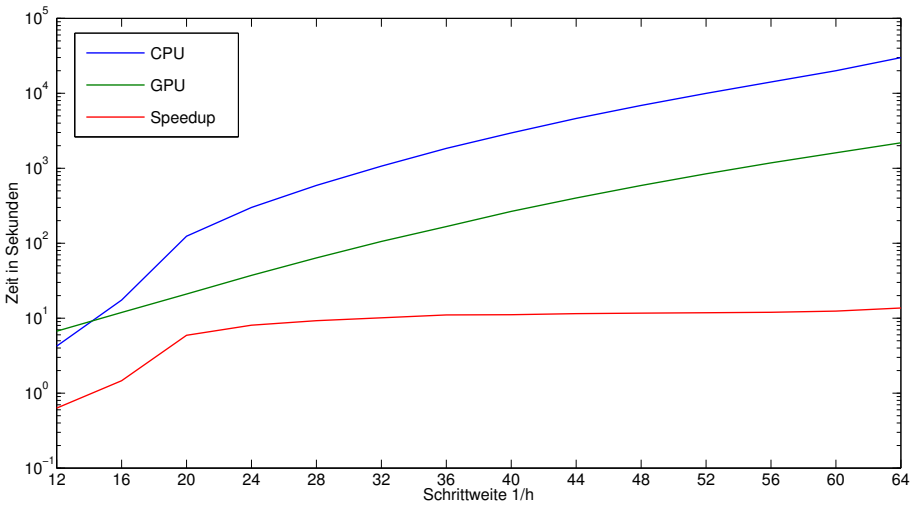


Abbildung 4.14: Laufzeitvergleich von CPU und GPU

4.3.3 Gegenüberstellung der Nummerierungstechniken

Als Nächstes vergleichen wir die Assemblierung im PDO- (siehe 2.4.1) und SDO-Format (siehe 2.4.2) und untersuchen, welchen Einfluss die beiden Nummerierungstechniken auf die zur Lösung des Problems benötigte Zeit haben. Um die Auswirkung der beiden Formate auf das Verhalten der Löser zu testen, haben wir zusätzlich die Matrix, die bei Benutzung der SDO-Nummerierung entsteht, im normalen DIA-Format gespeichert. Das zu lösende Problem ist weiterhin das in Kapitel 4.3 beschriebene.
Es ist leicht einzusehen, dass die Nummerierungsart keine Auswirkungen auf die Kondition der Systemmatrix haben kann. Die Matrizen, die bei der Assemblierung in den beiden Formaten entstehen, gehen, da sie sich nur in der Nummerierung der Unbekannten voneinander unterscheiden, durch die Multiplikation mit einer Permutationsmatrix P auseinander hervor. Diese ist naturgemäß regulär und hat die Determinante ± 1, was insbesondere bedeutet, dass sie die Konditionszahl nicht beeinflussen kann. Da diese aber maßgeblich für das Konvergenzverhalten unserer Löser ist, können sich die Iterationszahlen sowie die Defektverläufe für die beiden Nummerierungstechniken nicht unterscheiden. Lediglich bei der Verwendung der Gauß-Seidel und ILU(0)-Vorkonditionierer sind die beiden Formate aufgrund der Speicherung im DIA- Format und der unterschiedlichen Besetzungsstruktur der Matrizen nicht gleichwertig. Bezüglich der Laufzeit sind jedoch in jedem Fall Unterschiede festzustellen. Das Gleichungssystem im PDO-Format lässt sich schneller assemblieren und lösen, da die Systemmatrix für den betrachteten Testfall nur 69 Bänder hat. Im Gegensatz dazu besitzt die Matrix im SDO-Format 81 Bänder, was zum Beispiel bei Matrix-Vektor-Operationen zu einem erhöhten Rechenaufwand führt. Die Rechenzeiten und der Speedup sind den folgenden Abbildungen 4.15 und 4.16 zu entnehmen, wobei weiterhin das BiCGStab-Verfahren mit Jacobi-Vorkonditionierung verwendet wurde. In ersterer ist der CPU-, in letzterer der GPU-Zeitvergleich des SDO-Formats gegen das PDO-Format zu sehen. Dabei wurde auf der CPU parallel mit 6 Threads und auf der GPU mit 512 Threads pro Block gerechnet.

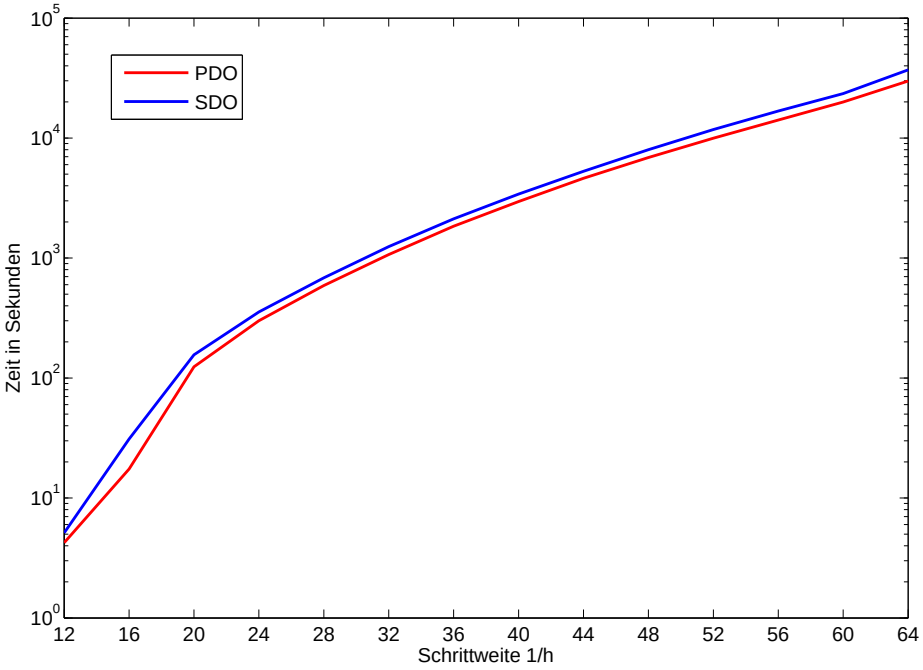


Abbildung 4.15: Zeitvergleich CPU

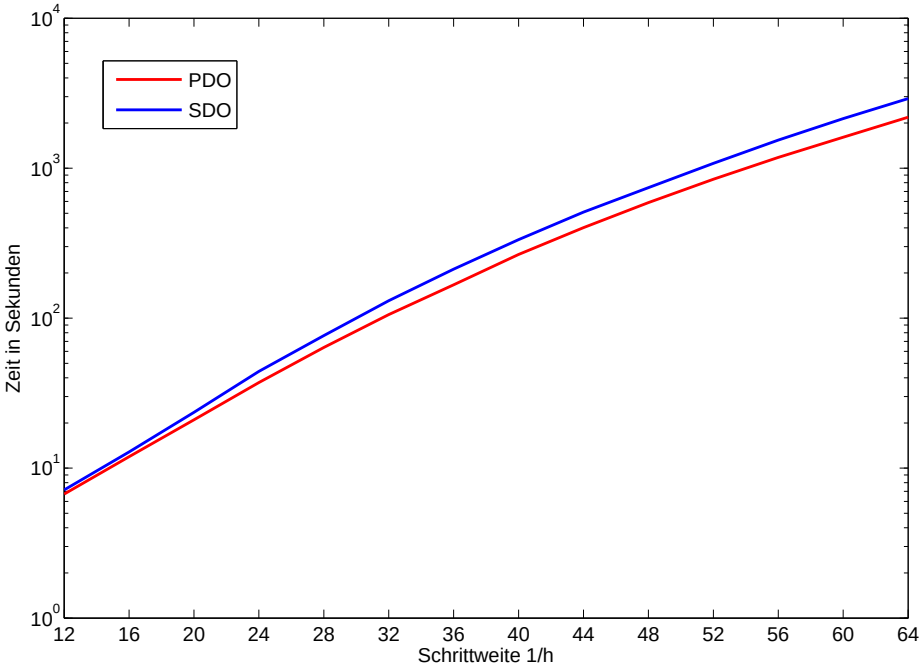


Abbildung 4.16: Zeitvergleich GPU

Zusammenfassend lässt sich sagen, dass die Verwendung der SDO-Nummerierung keinen Vorteil bringt, wenn die entstehende Blockstruktur nicht ausgenutzt wird. Die Frage, ob sich Letzteres auszahlt, wird im folgenden Kapitel beantwortet.

4.4 Löser mit konventionellen und Block-Vorkonditionierern

Der Testfall, mit dem wir uns in den folgenden Abschnitten beschäftigen, ist ein Zugversuch, der an einem Block aus Stahl durchgeführt wird (vgl. Abbildung 4.17). Die obere und untere Seitenfläche des Blockes ist in x- und y-Richtung fest eingespannt, das heißt sie können sich in diese Richtungen nicht verschieben. Zusätzlich erzwingen wir auf der oberen Seitenfläche mithilfe inhomogener Dirichlet-Randbedingungen eine Verschiebung von $u_{D_z} = 2e\text{-}11$ und auf der unteren Seitenfläche eine Verschiebung von $u_{D_z} = -2e\text{-}11$ in positive z-Richtung. Die Schrittweite des Gitters haben wir hier als $h = \frac{1}{32}$ und den Stellengewinn als $\texttt{epsilon} = 1e\text{-}12$ gewählt.

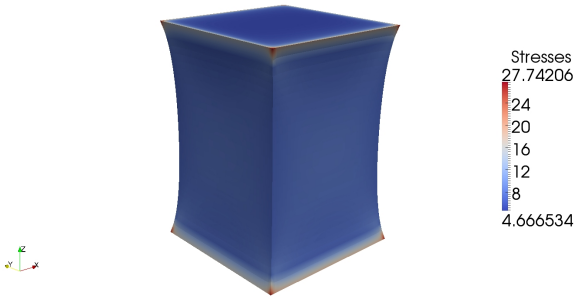


Abbildung 4.17: Zugversuch mit inhomogenen Dirichlet-Randbedingungen zu der Schrittweite $h = \frac{1}{32}$

4.4.1 Test verschiedener Löser und Block-Vorkonditionierer

Ordnet man die Unbekannten nach Kopplung, so erhält die Systemmatrix Blockgestalt (vgl. Kapitel 2.4). Beim Lösen verwenden wir deshalb Block-Vorkonditionierer und Löser, deren Eigenschaften wir im Folgenden untersuchen werden. Das Konzept dieser Löser und Block-Vorkonditionierer wurde in Kapitel 2.4.3 beschrieben und besteht im Wesentlichen darin, dass der Block-Vorkonditionierer wieder einen Löser aufruft, der seinerseits einen Vorkonditionierer benutzt. In den folgenden Tests wurde OpenMP zum parallelen Rechnen verwendet, die benutzte Hardware ermöglichte uns das Rechnen auf maximal 6 Kernen. Die Löserroutine ist dabei nicht vollständig parallelisiert, das heißt, dass die beiden Block-Vorkonditionierer und der Großteil der Vorkonditionierer nur seriell zu Verfügung stehen und nur die Löser parallel rechnen. Als Block-Vorkonditionierer können wir auf den Block-Gauß-Seidel- und den Block-Jacobi-Vorkonditionierer zurückgreifen. Bei den Lösern können wir, sowohl für das äußere Gleichungssystem als auch die inneren Probleme, jeweils zwischen dem BiCGStab- und dem Richardson-Verfahren wählen. Für den inneren Löser können wir die seriellen Vorkonditionierer, das heißt die Jacobi-, die Gauß-Seidel- oder die ILU(0)-Vorkonditionierung verwenden. Als einziger paralleler Vorkonditionierer steht uns der parallele Jacobi-Vorkonditionierer zur Verfügung.
Zunächst beschäftigen wir uns mit der Frage, wie man äußere Löser und Block-Vorkonditionierer bestmöglich miteinander kombinieren kann. Zu diesem Zweck wurden die inneren Löser und Vorkonditionierer und die dabei benötigten Parameter konstant gehalten. Als inneren Löser wählten wir das mit Gauß-Seidel vorkonditionierte BiCGStab-Verfahren mit einem Stellengewinn von $\texttt{epsilon} = 1e\text{-}12$ als Abbruchkriterium. Als Schrittweite für das zugrundeliegende Gitter haben wir $h = \frac{1}{32}$ gewählt. Die äußeren Löser und Block-Vorkonditionierer wurden variiert und die Iterationszahlen bis zu einem Stellengewinn von $\texttt{epsilon} = 1e\text{-}11$ notiert. Letztere sind in Abbildung 4.18 zu sehen.
Wie erwartet zeigt sich in dieser Abbildung, dass das BiCGStab-Verfahren besser konvergiert als das Richardson-Verfahren, sofern das Gleichungssystem unter Verwendung des gleichen Vorkonditionierers mit gleicher Genauigkeit gelöst wird. Wählt man außen den Block-Jacobi-Vorkonditionierer, so lässt sich beobachten, dass das BiCGStab-Verfahren nach 62 Iterationen die geforderte Genauigkeit erreicht, wohingegen das Richardson-Verfahren in dieser Situation 107 Iterationen benötigt. Des Weiteren lässt

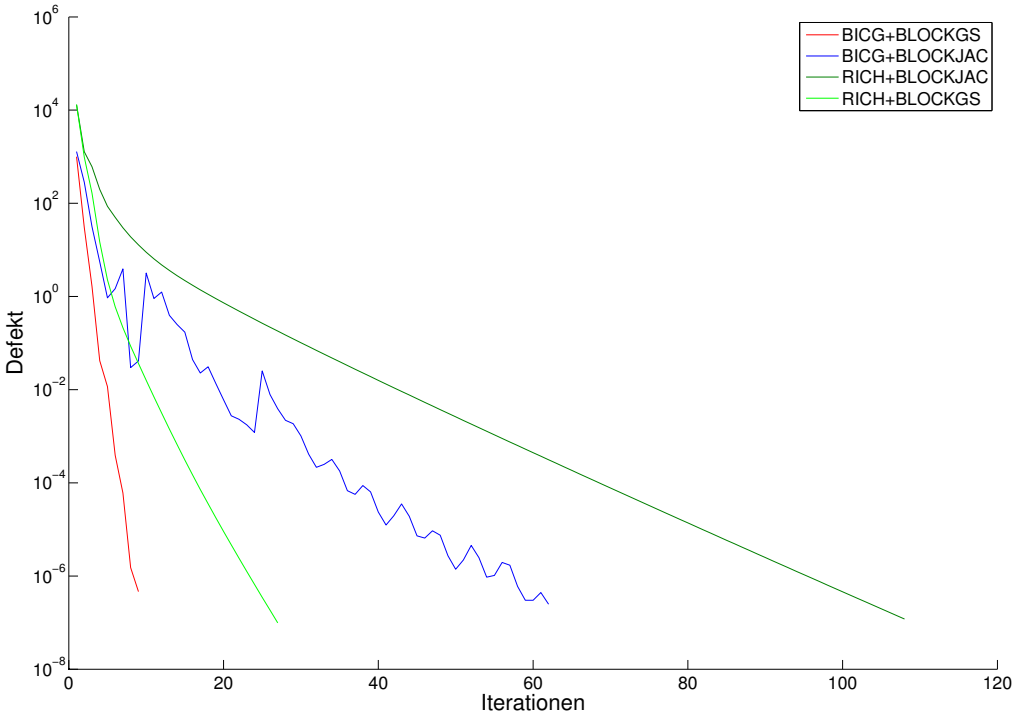


Abbildung 4.18: Konvergenzverlauf mit $h = \frac{1}{32}$

sich feststellen, dass der Block-Gauß-Seidel- dem Block-Jacobi-Vorkonditionierer vorzuziehen ist, da sowohl das BiCGStab- als auch das Richardson-Verfahren unter Verwendung des Block-Gauß-Seidel-Vorkonditionierers in weniger Iterationen den gewünschten Stellengewinn erzielen. Hier konvergiert das BiCGStab-Verfahren nach neun Iterationen, wohingegen das Richardson-Verfahren 26 Iterationen benötigt (vgl. Tabelle 4.3).

	Block-Jacobi	Block-Gauß-Seidel
Richardson-Verfahren	107	26
BiCGStab-Verfahren	62	9

Tabelle 4.3: Iterationszahlen

Außerdem lässt sich ein klarer Unterschied zwischen dem Block-Jacobi- und dem Block-Gauß-Seidel-Vorkonditionierer erkennen. Unabhängig von der Wahl des äußeren Lösers benötigt der Block-Gauß-Seidel-Vorkonditionierer weniger Iterationen als der Block-Jacobi-Vorkonditionierer. Dies sieht man daran, dass das Richardson-Verfahren mit Block-Gauß-Seidel-Vorkonditionierung 26 Iterationen benötigt und somit weniger als das BiCGStab-Verfahren mit dem Block-Jacobi-Vorkonditionierer, welches 62 Iterationen benötigt. Dies entspricht einer Verschlechterung um den Faktor 2.4.

	Block-Jacobi	Block-Gauß-Seidel
Richardson-Verfahren	643.78	147.43
BiCGStab-Verfahren	1037.90	125.72

Tabelle 4.4: Gesamtlaufzeit in Sekunden

Wie schon am Verhalten der Iterationszahlen erkennbar ist, hängt der zeitliche Vorteil des Block-Gauß-Seidel-Vorkonditionierers nicht von dem verwendeteten äußeren Löser ab (vgl. Tabelle 4.4). Der Block-Gauß-Seidel-Vorkonditionierer kombiniert also den Vorteil weniger Iterationen mit geringer Laufzeit und kompensiert dabei den erhöhten Rechenaufwand gegenüber dem Jacobi-Vorkonditionierer. Außerdem kann man aus diesen Ergebnissen folgern, dass man die kürzeste Gesamtlaufzeit des kompletten Lösungsvorgangs durch die Kombination der Gauß-Seidel-Vorkonditionierung mit dem BiCGStab-Verfahren erhält.

Nun wollen wir uns mit der Frage beschäftigen, welche Kombination aus Löser und Vorkonditionierer für das Lösen der inneren Gleichungssysteme am sinnvollsten ist. Auch in diesem Abschnitt lösen wir die inneren Gleichungssysteme mit einem Stellengewinn von $\texttt{epsilon} = 1e\text{-}12$ und wählen als feste Schrittweite für das zugrundeliegende Gitter $h = \frac{1}{32}$. Da wir aus dem vorherigen Abschnitt schon die Erkenntnis gewonnen haben, dass das BiCGStab-Verfahren mit dem Block-Gauß-Seidel-Vorkonditionierer die beste Kombination für das Lösen des äußeren Gleichungssystems ist, testen wir mit dieser Konfiguration die inneren Löser und Vorkonditionierer. Für diese haben wir das BiCGStab- und das Richardson-Verfahren in Kombination mit dem seriellen oder parallelen Jacobi-, dem Gauß-Seidel- oder dem ILU(0)-Vorkonditionierer zur Auswahl.

Die folgenden Tabellen zeigen die Anzahl der äußeren Iterationen (vgl. Tabelle 4.5), die Gesamtzahl der inneren Iterationen (vgl. 4.6), die der Restarts (vgl. 4.7) und die Gesamtlaufzeit des Lösers (vgl. 4.8). Die einzige Kombination, die divergiert, ist das Richardson-Verfahren mit dem seriellen/parallelen Jacobi-Vorkonditionierer. Ansonsten sind die äußeren Iterationen konstant neun.

	Jacobi (parallel)	Jacobi (seriell)	Gauß-Seidel	ILU(0)
BiCGStab-Verfahren	9	9	9	9
Richardson-Verfahren	divergent	divergent	9	9

Tabelle 4.5: Anzahl der äußeren Iterationen

	Jacobi (parallel)	Jacobi (seriell)	Gauß-Seidel	ILU(0)
BiCGStab-Verfahren	33482	33486	92297	10217
Richardson-Verfahren	divergent	divergent	1217532	363994

Tabelle 4.6: Gesamtzahl der inneren Iterationen

	Jacobi (parallel)	Jacobi (seriell)	Gauß-Seidel	ILU(0)
BiCGStab-Verfahren	3282	3149	47028	848

Tabelle 4.7: Gesamtzahl der inneren Restarts

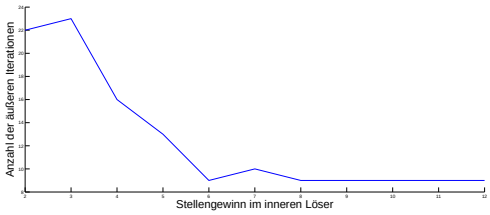
	Jacobi (parallel)	Jacobi (seriell)	Gauß-Seidel	ILU(0)
BiCGStab-Verfahren	14.05	16.06	125.72	31.86
Richardson-Verfahren	divergent	divergent	491.58	454.97

Tabelle 4.8: Gesamtlaufzeit des Lösers (in sek)

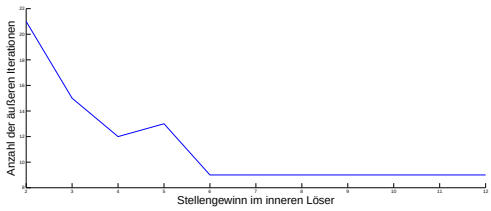
Auch in diesem Test zeigt sich, dass das BiCGStab-Verfahren für die inneren Gleichungssysteme besser geeignet ist als das Richardson-Verfahren, was sich anhand der Iterationszahlen und der Gesamtlaufzeit verdeutlichen lässt. Im Folgenden werden wir zunächst nur die seriellen Vorkonditionierer miteinander

vergleichen. Hier erkennt man, dass die theoretische Hierarchie der Vorkonditionierer - der ILU(0)- ist effektiver als der Gauß-Seidel- und dieser wiederum besser als der Jacobi-Vorkonditionierer (vgl. Skript [Göddeke]) - in der Praxis nicht immer zu beobachten ist .
Aufgrund von Restarts des BiCGStab-Verfahrens bei serieller Jacobi- (3149) beziehungsweise Gauß-Seidel-Vorkonditionierung (47028), braucht der Löser mit dem Gauß-Seidel-Vorkonditionierer mehr Iterationen (92297) als mit dem theoretisch schlechteren seriellen Jacobi-Vorkonditionierer (33486), was dazu führt, dass die Gesamtlaufzeit des Lösers mit dem Gauß-Seidel-Vorkonditionierer länger ist als mit dem seriellen Jacobi-Vorkonditionierer. Wenn man sich die Gesamtlaufzeit des Lösers ansieht, stellt man fest, dass der serielle Jacobi-Vorkonditionierer (16.06) aufgrund seines geringen Rechenaufwands sogar um den Faktor 2 schneller ist als der serielle ILU(0)-Vorkonditionierer (31.86), obwohl dieser ungefähr dreimal so wenig Iterationen (10217) und Restarts (848) benötigt. Das heißt, die besseren Konvergenzeigenschaften des ILU(0)-Vorkonditionierers reichen in den inneren Gleichungssystemen nicht aus, um den Nachteil des größeren Rechenaufwandes gegenüber dem Jacobi-Vorkonditionierer auszugleichen. Dieses unvorhergesehene Verhalten ist aktuell nicht erklärbar.
Aus diesen Ergebnissen kann man folgern, dass der serielle Jacobi-Vorkonditionierer den anderen seriellen Vorkonditionierern vorzuziehen ist und man die kürzeste Gesamtlaufzeit des kompletten Lösers in Kombination mit dem BiCGStab-Verfahren erhält. Wenn man zudem noch die Gesamtlaufzeit des parallelen Jacobi-Vorkonditionierers (14.05) mit der seriellen Jacobi-Variante (16.06) vergleicht, erkennt man wie erwartet einen leichten Vorteil bei der parallelen Implementierung.

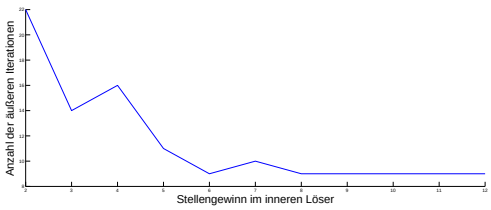
Als Fazit der letzten beiden Abschnitte kann man festhalten, dass zur schnellstmöglichen Lösung des Gleichungssystems bei Verwendung des Block-Formats die Kombination aus dem BiCGStab-Verfahren als äußerem Löser mit dem Block-Gauß-Seidel-Vorkonditionierer und dem mit Jacobi-vorkonditionierten BiCGStab-Verfahren als innerem Löser gewählt werden sollte. Bezüglich der Geamtlaufzeit des Lösers ist der parallele Jacobi-Vorkonditionierer für die inneren Gleichungssysteme die schnellste Wahl, weshalb wir auch im folgenden Test ausschließlich den parallelen Jacobi-Vorkonditionierer benutzen werden.



(a) $h = \frac{1}{16}$



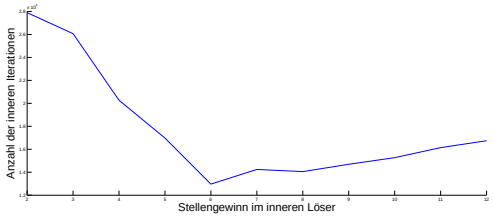
(b) $h = \frac{1}{32}$



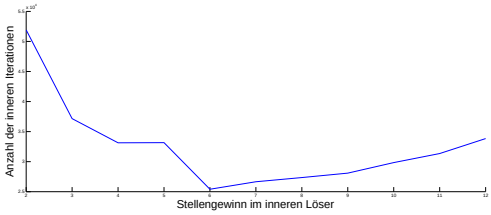
(c) $h = \frac{1}{64}$

Abbildung 4.19: Gesamtzahl der äußeren Iterationen zu verschiedenen Schrittweiten h

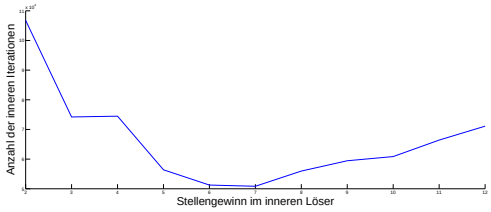
Im Folgenden werden wir, weiterhin am selben Testfall, untersuchen, wie genau man die inneren Gleichungssysteme lösen muss, um die äußeren und inneren Iterationen sowie die Laufzeit auf ein Minimum zu reduzieren. Wir haben dabei eine Konfiguration gewählt, die außen das BiCGStab-Verfahren mit dem Block-Gauß-Seidel-Vorkonditionierer und im Inneren das BiCGStab-Verfahren mit dem parallen Jacobi-Vorkonditionierer vorsieht. Getestet wurde mit den Schrittweiten $\frac{1}{16}$, $\frac{1}{32}$ und $\frac{1}{64}$, wobei der Stellengewinn im inneren Löser jeweils um den Faktor $\frac{1}{10}$ zwischen $\texttt{epsilon} = 1e\text{-}2$ und $\texttt{epsilon} = 1e\text{-}12$ variiert wurde. Die folgenden Graphen zeigen die Gesamtzahl der äußeren (vgl. Abb. 4.19) sowie der inneren Iterationen (vgl. Abb. 4.20) zu den obigen Schritweiten. Die Ergebnisse der Gesamtzeitmessungen sind in Abbildung 4.21 dargestellt.



(a) $h = \frac{1}{16}$

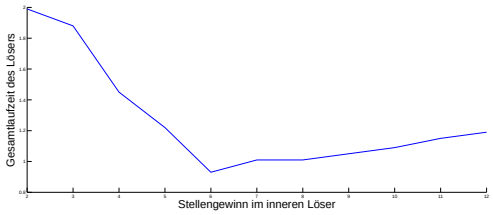


(b) $h = \frac{1}{32}$

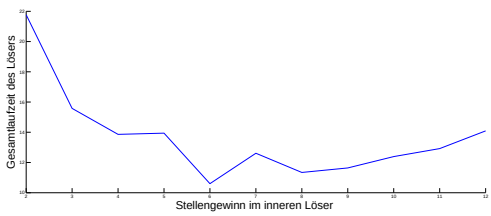


(c) $h = \frac{1}{64}$

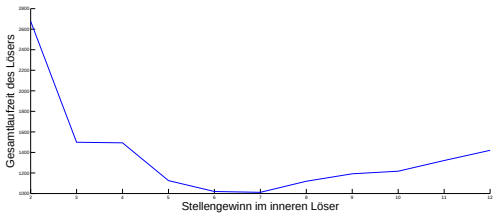
Abbildung 4.20: Gesamtzahl der inneren Iterationen zu verschiedenen Schritweiten h



(a) $h = \frac{1}{16}$



(b) $h = \frac{1}{32}$



(c) $h = \frac{1}{64}$

Abbildung 4.21: Gesamtrechendauer zu verschiedenen Schrittweiten h

Unsere Tests zeigen, dass für den inneren Löser ein Stellengewinn von $\texttt{epsilon} = 1e\text{-}6$ zu optimalen Ergebnissen führt. Ein weiterer Stellengewinn beim Lösen der inneren Gleichungssysteme führt zu keiner weiteren Verbesserung der äußeren Iterationszahlen. Die Theorie besagt, dass man die inneren Gleichungssysteme nicht exakt lösen muss, um Konvergenz zu erhalten [Axelsson], unser Test zeigt , dass der äußere Löser minimal neun Iterationen benötigt und diese Zahl schon bei einem Stellengewinn von $\texttt{epsilon} = 1e\text{-}6$ erreicht wird. Der äußere Löser konvergiert sogar bei geringerer Genauigkeit, was sich in mehr äußeren Iterationen niederschlägt und in Folge dessen zu einer längeren Laufzeit führt. Löst man die inneren Gleichungssysteme mit einer höheren Genauigkeit, so führt dies zu einem Mehraufwand beim Anwenden der inneren Löser, was wiederum einen Anstieg der inneren Iterationen und daher eine längere Laufzeit zur Folge hat. Die Theorie, dass sich die Iterationszahlen bei der Halbierung der Schrittweite verdoppeln, ist für die inneren Gleichungssysteme jedoch gültig, man vergleiche hierzu Tabelle 4.9. Gerechnet wurde hier mit dem optimalen Stellengewinn von $\texttt{epsilon} = 1e\text{-}6$.

	Gesamtlaufzeit(in sek)	innere Iterationen	innere Restarts	äußeren Iterationen
$h = \frac{1}{16}$	0.93	12966	1586	9
$h = \frac{1}{32}$	10.60	25408	2382	9
$h = \frac{1}{64}$	1019.69	51246	4085	9

Tabelle 4.9: Messergebnisse für $\texttt{epsilon} = 1e\text{-}6$

4.4.2 Vergleich zwischen normalem und Block-Löser bei SDO-Nummerierung

In diesem Abschnitt untersuchen wir die Güte der in Kapitel 2.4 vorgestellten Löser und Vorkonditionierer im Hinblick auf ihre Anwendung auf das Gleichungssystem, welches durch SDO-Nummerierung entsteht, und vergleichen dabei das konventionelle mit dem Block-Lösungsverfahren wie es in Kapitel 4.4.1 untersucht wurde. Anschließend verdeutlichen wir die Geschwindigkeitsdiskrepanz zwischen dem normalen und dem Block-Lösungsverfahren, um Rückschlüsse ziehen zu können, ob sich bei den uns zur Verfügung stehenden Konfigurationen für Block-Lösungsverfahren der Mehraufwand eines solchen lohnt.

Der betrachtete Testfall ist weiterhin die in Kapitel 4.4.1 beschriebene Zugprobe an einem stählernen Block (vgl. Abb. 4.17) und als Schrittweite für das zugrundeliegende Gitter haben wir $h = \frac{1}{32}$ gewählt. Wir verwenden auch hier OpenMP, um auf sechs Kernen parallel zu rechnen, wobei der Jacobi-Vorkonditionierer und die jeweiligen Löser parallel implementiert wurden. Als Erstes beschäftigen wir uns mit dem normalen Lösungsverfahren, um herauszufinden, wie sich die verschiedenen Konfigurationen aus Löser und Vorkonditionierer auf die Gesamtlaufzeit des Lösers und die Iterationszahlen auswirken. Die zu vergleichenden Kombinationen ergeben sich aus den zur Verfügung stehenden ILU(0)-, Gauß-Seidel- und dem parallelen Jacobi-Vorkonditionierer sowie dem BiCGStab- beziehungsweise Richardson-Verfahren als Löser. Die Ergebnisse der Tests sind in den folgenden Tabellen mit den Laufzeiten (vgl. Tabelle 4.10) und den Iterationszahlen (vgl. Tabelle 4.11) der verschiedenen Löser-Vorkonditionierer-Konfigurationen zu sehen. Abbildung 4.22 stellt die Resultate graphisch dar.

	Jacobi (parallel)	Gauß-Seidel	ILU(0)
Richardson-Verfahren	divergent	453.329	71.34
BiCGStab-Verfahren	638.07	108.976	5.09

Tabelle 4.10: Gesamtlaufzeit (in sek) des normalen Lösungsverfahren

	Jacobi (parallel)	Gauß-Seidel	ILU(0)
Richardson-Verfahren	divergent	15953	4755
BiCGStab-Verfahren	14844	1266	145

Tabelle 4.11: Anzahl der Iterationen des normalen Lösungsverfahren

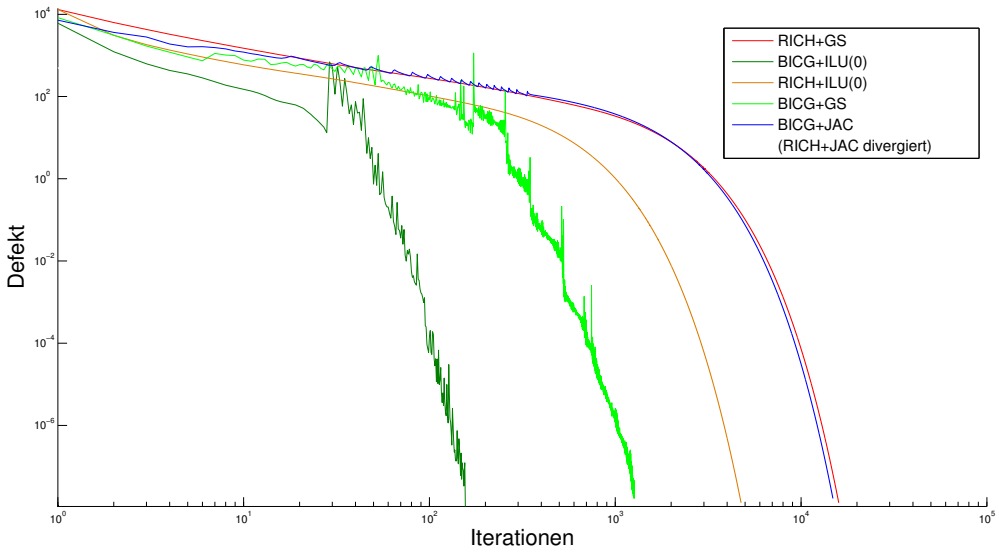


Abbildung 4.22: Konvergenzverlauf bei $h = \frac{1}{32}$

Bei den konventionellen Lösungsverfahren bestätigt sich die Theorie des Konvergenzverhaltens der Löser und Vorkonditionierer (vgl. Skript [Göddeke]). Der ILU(0)-Vorkonditionierer ist der mit Abstand effektivste von uns getestete Vorkonditionierer, der parallele Jacobi-Vorkonditionierer der ineffektivste und das BiCGStab-Verfahren ist weitaus schneller als das Richardson-Verfahren. Besonders deutlich wird dies anhand der Tatsache, dass die Kombination aus Richardson-Löser und Gauß-Seidel-Vorkonditionierer circa 16000 Iterationen und etwa 7.5 Minuten, die Kombination aus BiCGStab-Löser und parallelem Jacobi-Vorkonditionierer hingegen ungefähr 15000 Iterationen und über 10 Minuten benötigt. Dies bedeutet, dass der bessere Vorkonditionierer sogar das asympthotisch schlechtere Konvergenzverhalten des Richardson-Verfahrens kompensiert. Vergleicht man die beiden Löser, die jeweils den Gauß-Seidel-Vorkonditionierer verwenden, so ist das BiCGStab-Verfahren um einen Faktor vier besser. Die beste Kombination ergibt sich aus dem BiCGStab-Verfahren mit dem ILU(0)-Vorkonditionierer, welche nur 145 Iterationen und 5.9 Sekunden benötigt. Dies entspricht einer Steigerung um den Faktor 125 gegenüber der Laufzeit des BiCGStab-Verfahrens mit dem parallelen Jacobi-Vorkonditionierer beziehungsweise einem Faktor 21 gegenüber der Laufzeit des BiCGStab-Verfahrens mit dem Gauß-Seidel-Vorkonditionierer. Außerdem ergibt sich beim Richardson-Verfahren vom Gauß-Seidel- zum ILU(0)-Vorkonditionierer eine Verbesserung der Iterationen um den Faktor 3.3. Beim BiCGStab-Verfahren ergibt sich sogar eine Verbesserung um den Faktor 8.7. Die beste Kombination bei den normalen Lösungsverfahren ist, bei den uns zur Verfügung stehenden Vorkonditionierern und Lösern, also das BiCGStab-Verfahren mit dem ILU(0)-Vorkonditionierer, was aufgrund der Theorie auch zu erwarten war.

Im folgenden Abschnitt wollen wir das normale und das Block-Lösungsverfahren miteinander vergleichen. Dazu dienen uns die Ergebnisse aus dem vorherigen Abschnitt über die normalen Lösungsverfahren mit Schrittweite $h = \frac{1}{32}$ und die Resultate aus Kapitel 4.4.1 bezüglich der Block-Lösungsverfahren. Da sich das Block-Lösungsverfahren aus zwei potentiellen Vorkonditionierern und zwei Lösern zu-

sammensetzt, die die Rechenzeit unterschiedlich beeinflussen, kann man dieses nicht eins zu eins mit dem normalen Lösungsverfahren vergleichen, welches nur aus jeweils einem Löser und einem Vorkonditionierer besteht. Deshalb müssen wir die Ergebnisse etwas differenzierter betrachten. Als beste Konfiguration für das Block-Lösungsverfahren benutzen wir, wie in Kapitel 4.4.1 herausgefunden, das BiCGStab-Verfahren als äußeren Löser mit dem Block-Gauß-Seidel-Vorkonditionierer sowie das mit Jacobi-vorkonditionierte BiCGStab-Verfahren als inneren Löser mit einem Stellengewinn von `epsilon` $= 1e$-6. Zum Vergleich dient die Tabelle 4.12, die die Gesamtlaufzeiten der jeweiligen Lösungsmethoden gegenüberstellt.

	Gesamtlaufzeit des Lösers (in sek)	Anzahl der Iterationen
normale Lösungsverfahren:		
BiCGStab-Jacobi (parallel)	638.07	14844
BiCGStab-Gauß-Seidel	108.976	1266
BiCGStab-ILU(0)	5.09	145
optimales Block-Lösungsverfahren	10.60	25408 (innere Iter.)

Tabelle 4.12: Vergleich der verschiedenen Löserkombination für $h = \frac{1}{32}$

Wenn man diese Ergebnisse vergleicht, sieht man, dass die obige Konfiguration des Block-Lösungsverfahrens (10.60 sek) den beiden schwächeren normalen Lösungsverfahren (638.07 sek, 108.976 sek) überlegen ist. Lediglich der ILU(0)-Vorkonditionierer (5.09 sek) ist schneller als die optimale Block-Variante. Um einen besseren Eindruck zu bekommen, haben wir die beiden Konfigurationen für die Schrittweite $h = \frac{1}{64}$ miteinander verglichen (vgl. Tabelle 4.13).

	Gesamtlaufzeit des Lösers (in sek)	Anzahl der Iterationen
normale Lösungsverfahren:		
BiCGStab-ILU(0)	120.25	272
BiCGStab-Gauß-Seidel	3623.99	5187
optimales Block-Lösungsverfahren	1019.69	51246 (innere Iter.)

Tabelle 4.13: Vergleich der besten Löserkombination für $h = \frac{1}{64}$

Es ergibt sich hier, dass das optimale Block-Lösungsverfahren (1019.69 sek) um den Faktor drei schneller ist als das normale Lösungsverfahren mit der Gauß-Seidel-Vorkonditionierung (3623.99 sek). Vergleicht man dieses hingegen mit dem normalen Lösungsverfahren, das den ILU(0)-Vorkonditionierer verwendet (120.25 sek), erkennt man, dass letzteres deutlich schneller ist als das Block-Lösungsverfahren, was auch schon für die Schrittweite $h = \frac{1}{32}$ zu beobachten war (vgl. Tabelle 4.12).

Als Fazit lässt sich festhalten, dass sich das Ausnutzen der Block-Struktur im Hinblick auf die Gesamtlaufzeit auszahlen kann. Dies kann aber nicht verallgemeinert werden, weil die Gesamtlaufzeit des Block-Lösers stark von der Wahl des äußeren Block-Vorkonditionierers abhängt. Wenn man die uns zur Verfügung stehenden Konfigurationen der normalen beziehungsweise Block-Lösungsverfahren betrachtet, ist das schnellste Verfahren zur Lösung des Gleichungssystems bei SDO-Nummerierung das normale Lösungsverfahren bestehend aus dem BiCGStab-Verfahren mit dem ILU(0)-Vorkonditionierer. Man muss jedoch berücksichtigen, dass wir keinen starken Block-Vorkonditionierer zur Verfügung haben und deshalb der Vergleich mit dem normalen Lösungsverfahren bei ILU(0)-Vorkonditionierung nicht representativ ist. Legt man zum Vergleich stattdessen das normale Lösungsverfahren mit Gauß-Seidel-Vorkonditionierung zugrunde, so ist das optimale Block-Lösungsverfahren bestehend aus dem BiCGStab-Verfahren als äußerem Löser mit dem Block-Gauß-Seidel-Vorkonditionierer und dem mit Jacobi vorkonditionierten BiCGStab-Verfahren als innerem Löser mit einem Stellengewinn von `epsilon` $= 1e$-6 die schnellere Wahl.

Kapitel 5

Showcases

In dem folgenden Abschnitt werden wir anhand zweier komplizierter Testfälle demonstrieren, dass unsere Programme auch zur Lösung von Problemen eingesetzt werden können, die weit über die einfachen Zug- und Druckproben der vorherigen Kapitel hinausgehen. Als Testobjekt wird uns hier eine „virtuelle Voodoo-Puppe“ aus Stahl ($E = 2.1 \cdot 10^{11}$ Pa, $\nu = 0.29$) dienen, die wir diversen Belastungen aussetzen. Die zugehörige Geometrie ist in Abbildung 5.1 zu sehen. Zur Lösung haben wir den BICGStab- Löser mit der ILU(0)-Vorkonditionierung eingesetzt, also die Kombination, die sich in den Tests aus Kapitel 4.4 als optimal herausgestellt hat. Als Stellengewinn haben wir `epsilon` $= 10^{-11}$, als Startlösung $x_0 = (10^{-9})_{i=1,\dots,N}$ verwendet. Bei der Assemblierung haben wir erneut die PDO-Nummerierung (vgl. Kapitel 2.4) benutzt. Man beachte, dass wir auch bei diesem Testfall gemäß des Ansatzes der Immersed-Boundary-Methoden (vgl. Kapitel 2.3) immer auf dem gesamten Einheitswürfel rechnen. Die Simulation der Belastungen im Falle der komplizierten Geometrie wird ausschließlich durch nachträgliche Manipulationen an der Systemmatrix ermöglicht. Im Detail bedeutet dies, dass auch bei den folgenden Szenarien immer erst die Systemmatrix zu einem rundherum mit Neumann-Rändern versehenen Einheitsblock assembliert wurde und erst dann durch zusätzliche Neumann-Randbedingungen die Kontur der Voodoo-Puppe von dem Rest des Blockes gelöst wurde. Die Punkte des Würfels, die außerhalb des betrachteten Volumens liegen, haben wir dabei mit homogenen Dirichlet-Randbedingungen versehen und bei der graphischen Ausgabe vernachlässigt.

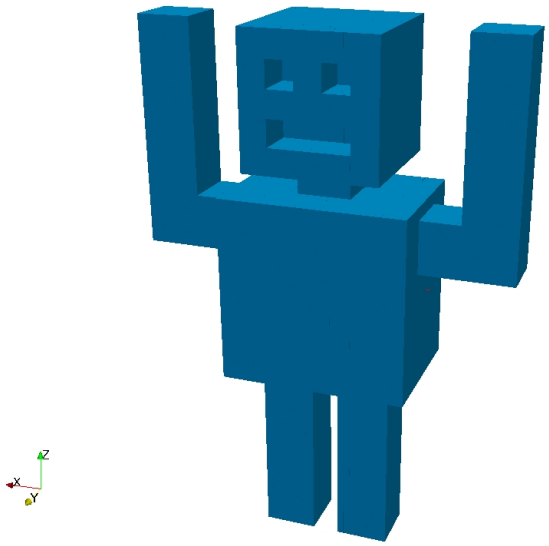
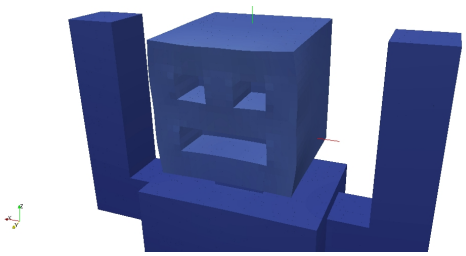
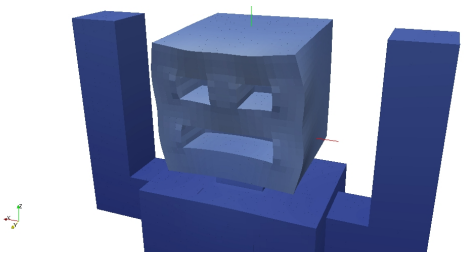


Abbildung 5.1: Geometrie der Voodoo-Puppe bei $h = \frac{1}{80}$

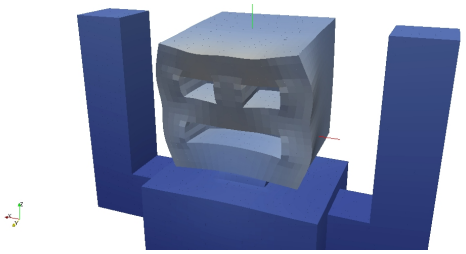
In den Abbildungen 5.2 a) bis f) sind nun die Ergebnisse einiger Berechnungen zu sehen, die wir für den „Voodoo-Puppen“-Testfall durchgeführt haben. Simuliert wurden hier die Konsequenzen einer äußeren Krafteinwirkung t auf die vordere Hälfte der Kopfoberseite. Die Beine wurden zu diesem Zweck durch homogene Dirichlet-Randbedingungen fixiert, die durch semi-implizite Integration in der Systemmatrix verarbeitet wurden.



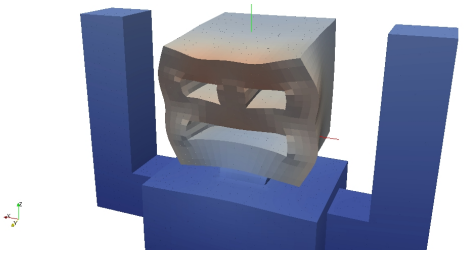
(a) Belastung durch t =500Pa



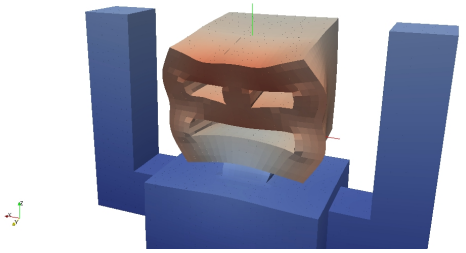
(b) Belastung durch t =1000Pa



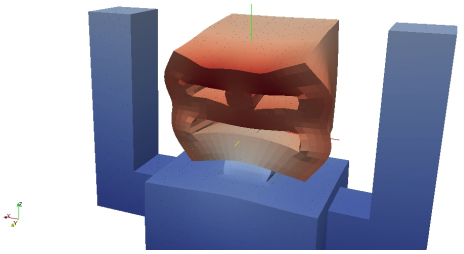
(c) Belastung durch t =1500Pa



(d) Belastung durch t =2000Pa



(e) Belastung durch t =2500Pa



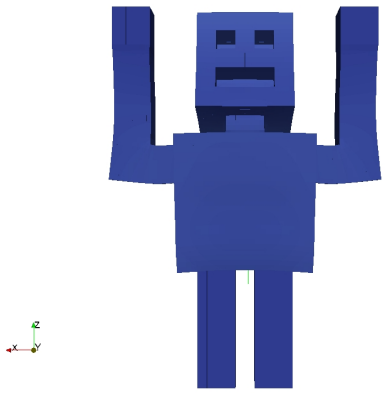
(f) Belastung durch t =3000Pa

Abbildung 5.2: Simulation der Auswirkungen verschiedener äußerer Krafteinwirkungen auf die Kopfoberseite (in negative z-Richtung). Skaliert wurde hier mit dem Faktor 10^7, die Färbungen sind ein Maß für die Stärke der lokalen Verschiebung. Gerechnet wurde auf einem Gitter der Schrittweite $h = \frac{1}{80}$.

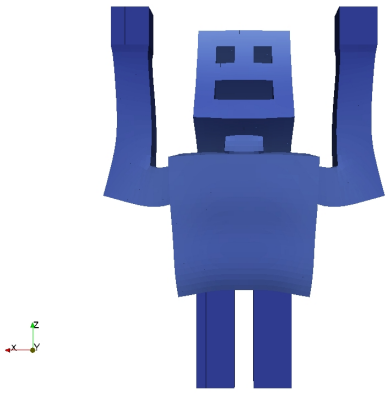
Man erkennt, dass das Gesicht der Voodoo-Puppe ein ähnliches Verhalten zeigt wie der Block im Testfall des Kapitels 4.3. Ein wesentlicher Unterschied besteht hier jedoch darin, dass die Löcher im Kopf nicht durchgängig sind (vgl. Abbildung 5.1). Neben der Wirkung einer äußeren Belastung lassen sich anhand der Voodoo-Puppe auch die Folgen einer internen Krafteinwirkung simulieren. Eine solche haben wir bisher nur in Kapitel 4.1 zur Herleitung einer analytischen Referenzlösung genutzt, wobei wir dort lediglich einen in allen Oberflächenpunkten fixierten Block betrachtet haben. Zur Ergänzung werden wir hier abschließend noch ein Beispiel dafür geben, wie sich diese internen Lasten in Verbindung mit freien Neumann-Rändern auswirken. Dazu werden wir eine ortsabhängige Volumenkraftdichte der Form

$$f(x, y, z) = -c(1 - z)^2 \cdot e_z$$

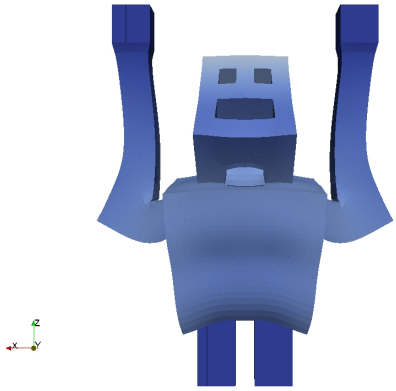
in negative z-Richtung an der Voodoo-Puppe angreifen lassen, wobei wir die Hände und Beine erneut durch semi-implizit integrierte, homogene Dirichlet-Randbedingungen fixieren. Die Ergebnisse dieser Belastung zu verschiedenen c-Werten sind in den Abbildungen 5.3 a) bis d) zu sehen.



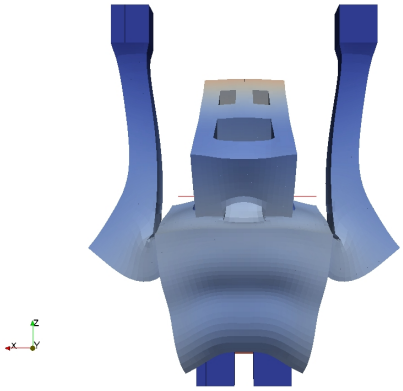
(a) Verformung bei $c = 100000$



(b) Verformung bei $c = 250000$



(c) Verformung bei $c = 500000$



(d) Verformung bei $c = 750000$

Abbildung 5.3: Deformationen in Folge der oben beschriebenen internen Krafteinwirkung f zu verschiedenen Werten des Parameters c. Die Verschiebungen wurden hier mit dem Faktor 10^7 skaliert, die Färbung spiegelt die Stärke der lokalen Verformung wider. Die verwendete Schrittweite war $h = \frac{1}{80}$.

Betrachtet man die Iterationszahlen und Laufzeiten für die beiden Testreihen, so stellt man fest, dass sich in den beiden Belastungssituationen ein ähnliches Konvergenzverhalten einstellt, man vergleiche hierzu die Tabellen 5.1 und 5.2. Zudem ist hier deutlich zu erkennen, dass die Iterationszahlen in keinem erkennbaren Zusammenhang zu den beiden Parametern t und c stehen. Das Verhalten der iterativen Löser ist somit primär von der Lagersituation, nicht aber von der konkreten Höhe der Belastungen abhängig.

Belastung t	Iterationszahl	Laufzeit (in Sekunden)	Konvergenzrate
500Pa	2145	942.90	0.9882
1000Pa	2281	1000.80	0.9889
1500Pa	2537	1111.34	0.9900
2000Pa	4268	1891.76	0.9941
2500Pa	2556	1118.14	0.9900
3000Pa	3038	1329.76	0.9917

Tabelle 5.1: Benötigte Iterationen, Laufzeiten und Konvergenzraten für die Testfälle aus Abbildung 5.2.

Parameter c	Iterationszahl	Laufzeit (in Sekunden)	Konvergenzrate
100000	2767	1208.46	0.9909
250000	2608	1145.30	0.9903
500000	1900	826.97	0.9867
750000	2519	1098.42	0.9902

Tabelle 5.2: Benötigte Iterationen, Laufzeiten und Konvergenzraten für die Testfälle aus Abbildung 5.3.

Kapitel 6

Fazit

Abschließend können wir zu dem Ergebnis kommen, dass sich die Laufzeiteffizienz unseres Simulationscodes insbesondere durch den Einsatz starker Vorkonditionierer und die Parallelisierung der Programmabläufe steigern lässt. Die Block-Lösungsverfahren sind hierzu zwar auch in der Lage, gegenüber des ILU(0)-Vorkonditionierers erweisen sie sich aber als nicht konkurrenzfähig. Das Ausnutzen der Blockstruktur der Systemmatrix wird nur dann interessant, wenn dieses Vorgehen eine Parallelisierung des Lösungsprozesses ermöglicht, die die Vorteile der im Allgemeinen nur schlecht parallelisierbaren, starken Vorkonditionierer aufwiegt. Bei unseren Tests konnten wir ein solches Verhalten aber nicht beobachten. Zu den von uns benutzten Parallelisierungsmethoden ist zu sagen, dass sich die GPU-Variante - zumindest bei Verwendung des parallelisierten Jacobi-Vorkonditionierers - als der OpenMP-Variante überlegen erwiesen hat. Der zweistellige Speedup spricht hier für sich. Man muss jedoch beachten, dass sich die Möglichkeiten der GPU nur unter hohem Programmieraufwand voll ausnutzen lassen. Stärkere Vorkonditionierer lassen sich hier nicht problemlos von der CPU auf die GPU übertragen, sondern müssen so abgeändert und manipuliert werden, dass kein Parallelisierungspotential verschenkt wird. Zur ökonomischen Programmentwicklung stellt die Nutzung von OpenMP somit trotz des niedrigeren Speedups eine vernünftige Alternative dar. Bezüglich der Immersed-Boundary-Methoden kommen wir zusammenfassend zu dem Urteil, dass das Penalty-Verfahren zwar funktionsfähig, aber im Allgemeinen nur schwer kontrollierbar ist. Die semi-implizite Integrationsmethode ist hier das Mittel der Wahl.

Diese rein sachlichen Ergebnisse sind jedoch nur sekundär von Bedeutung. Weit wichtiger sind die Erfahrungen, die wir im Laufe dieses Studienprojektes sammeln konnten. So war es uns aus erster Hand möglich zu erleben, wie sich die Codeentwicklung in einer größeren Gruppe gestaltet und welche Probleme beim arbeitsteiligen Programmieren entstehen können. Insbesondere auf dem Gebiet der modularen und flexiblen Programmentwicklung konnten wir hier unsere Kenntnisse ausbauen. Vorteilhaft war dabei, dass uns größtmögliche Freiheit in Bezug auf die Projektplanung und die explizite Realisierung der Programmiervorhaben gewährt wurde. Dadurch wurde uns zum einen erlaubt, die Aufgaben entsprechend der persönlichen Interessen und Stärken zu verteilen, und zum anderen, mit verschiedenen Ansätzen zu experimentieren und Tests auf vielen Gebieten durchzuführen. Rein fachlich konnten wir so unsere Kenntnisse über die Programmiersprache C, die Nutzung externer Bibliotheken, das Programmieren auf GPUs, die Diskretisierung mit Finiten Differenzen und die Freiheitsgradverwaltung ausbauen und kamen zusätzlich in Kontakt mit neuen Vorkonditionierungs- und Lösungstechniken. Im Großen und Ganzen stellte dieses Studienprojekt eine erfreuliche Abwechslung und eine sinnvolle, praxisorientierte Erweiterung des übrigen Technomathematik-Studiums dar und wird uns deshalb als bereichernde Erfahrung in Erinnerung bleiben.

Literaturverzeichnis

[Göddeke] D. Göddeke, Vorlesungsskript zu High Performance Computing und Parallele Numerik, TU Dortmund, SS 2011

[Bartel] T. Bartel, Mitschrift von Niklas Borg der Vorlesung Mechanik II, Fakultät Maschinenbau, Sommersemester 2011

[Wobker] H. Wobker, Efficient Multilevel Solvers and High Performance Computing Techniques for the Finite Element Simulation of Large-Scale Elasticity Problems, Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften, Fakultät für Mathematik, TU Dortmund, 2009

[Kessel] S. Kessel, Skriptum zur Vorlesung MECHANIK II für Studierende des Maschinenbaus, Fakultät Maschinenbau, Lehrstuhl für Mechanik, TU Dortmund, 1997

[Halder] S. Halder, Frequenzfilternde Zerlegungen zur numerischen Lösung der Stokes-Gleichungen, Diplomarbeit am Institut für Computeranwendungen III, Universität Stuttgart, 1998

[Axelsson] O. Axelsson; On iterative solvers in tructural mechanics; separate displacement ordering and mixed variable methods, Mathematics and Computers in Simulation 1999; 50:11-30

[Mittal] R. Mittal und G. Iaccarino, Immersed boundary methods, Annu. Rev. Fluid Mech., 37:239-261, 2005, verfügbar unter `http://www.annualreviews.org/doi/abs/10.1146/annurev.fluid.37.061903.175743`.

[Königsberger] K. Königsberger, Analysis 1, Springer-Verlag, 6. Auflage, 2003

[Möller] M. Möller, Skriptum Discretization Techniques, TU Dortmund, SS 2011

[Knabner] P. Knabner, L. Angermann, Numerik partieller Differentialgleichungen, Springer-Verlag, 2000

[NVIDIA] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide, Version 1.1, 29.11.2007, verfügbar unter `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`

[Garland] M. Garland and D. B. Kirk, Understanding throughput-oriented architectures, Commununications of the ACM, 2010; 53:58-66

[Roos] H.-G. Roos und C. Großmann, Numerische Behandlung partieller Differentialgleichungen, Teubner Verlag, 2005

[Dongarra] J. Dongarra et al, The International Exascale Software Project roadmap, International Journal of High Performance Computing Applications, 2011

Abbildungsverzeichnis

Tabellenverzeichnis