FRIEDRICH-ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl 10 Systemsimulation

Effizientes Berechnen lokaler Steifigkeitsmatrizen

Robert Kurin

Bachelorarbeit

Effizientes Berechnen lokaler Steifigkeitsmatrizen

Robert Kurin

Bachelorarbeit

Aufgabensteller:	Prof. Dr. Christoph Pflaum
Betreuer:	Prof. Dr. Christoph Pflaum
Bearbeitungszeitraum:	20.01.2023 - 20.06.2023

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22. Mai 2023 .....

Inhaltsverzeichnis

1 Kurzfassung

In dieser Bachelorarbeit wird die Berechnung lokaler Steifigkeitsmatrizen im Kontext der Finiten Elemente Methode, insbesondere zur Lösung der Poisson-Gleichung, im Detail beschrieben und mittels einer eigenen C++ Implementierung umgesetzt. Die Zielsetzung ist hierbei, im Rahmen einer Diskretisierung und globalen Steifigkeitsmatrixberechnung durch das Programm UGBlocks [Pfl10] ein möglichst laufzeiteffizientes Programm zur Berechnung der zur Lösung der Poisson-Gleichung benötigten lokalen Matrizen für lineare 2D-Elemente sowie lineare 3D-Elemente zu schreiben. Hierfür wurde im ersten Teil dieser Bachelorarbeit zunächst das benötigte Wissen zur Berechnung dieser Matrizen für die diversen Elementtypen recherchiert und ausgearbeitet, und anschließend in Form eines C++ Programms umgesetzt. Anschließend wurden mehrere Ansätze verfolgt, das Programm anzupassen, um seine Laufzeit zu verbessern. Hierbei konnte im Vergleich mit der ersten Implementierung eine Verkürzung der Rechenzeit von ca. 13 Prozent erreicht werden. Zur Prüfung der Korrektheit der Berechnung der Matrizen wurden die Ergebnisse des Programms mit denen des bereits in Kombination mit UGBlocks genutzten Programms Colsamm [Här07] verglichen. Hierbei wurde auch ein Laufzeitvergleich mit dieser auf Fast Expression Templates [Ale07] basierenden Implementierung vorgenommen. Colsamm war selbst nach den erreichten Laufzeitverbesserungen ca. 22 Prozent performanter als das selbstgeschriebene Programm. Zusätzlich wurde experimentell abgeschätzt, wie sehr sich die kombinierte Berechnungsdauer beider zum Aufstellen des zur Lösung des Poisson-problems nötigen Matrizen, sowohl für die eigene Implementierung als auch für Colsamm, durch ein simultanes Berechnen beider Matrizen für jedes Element der von UGBlocks gegebenen Diskretisierung verkürzen ließe. Hier ließ sich eine potentielle Laufzeitverbesserung von 17 bis 20 Prozent messen. Zuletzt wurde analysiert, für Probleme welcher Größenordnungen die Effizienz der lokalen Steifigkeitsmatrixberechnung den größten prozentualen Einfluss auf die Gesamtlaufzeit des Finiten Elemente Codes hat. Hierbei ergab sich, dass die Dauer der Berechnung der lokalen Steifigkeitsmatrizen für Diskretisierungen mit wenig Gitterpunkten in einem deutlich höheren Verhältnis zur Dauer des Lösens des entstehenden Gleichungssystems steht, als bei Diskretisierungen mit mehr Gitterpunkten.

2 Einführung

Das Lösen komplexer Differentialgleichungen ist in vielen Industriebereichen eine Notwendigkeit. Die Berechnung von mechanischen Verformungen, von Strömungen, Lasern, Temperaturverteilungen und vielen weiteren physikalischen Phänomenen hängen allesamt von hinreichend genauen und dennoch effizienten Methoden ab. Auch wenn die Leistungsfähigkeit der hierfür genutzten Computer in den letzten Jahrzehnten durchgängig gestiegen ist und nach wie vor immer weiter zunimmt, so sind auch die Ansprüche gestiegen, immer komplexere Probleme immer exakter zu lösen. Somit ist die Effizienz der genutzten Programme weiterhin ein relevanter Aspekt im Bezug auf Wirtschaftlichkeit und Performanz wissenschaftlicher und industrieller Berechnungen. Durch effizientere Programme können Rechenzeit, Anschaffungskosten für Hardware, und Stromkosten gespart werden. Ein Verfahren, welches seinen Ursprung in der Festkörpermechanik hat und seitdem in vielen industriellen und wissenschaftlichen Bereichen weitläufig genutzt wird, ist die Methode der Finiten Elemente [Rei19]. Sie dient dem Finden einer Annäherungslösung für Differentialgleichungen auf einem Gebiet, wobei das Gebiet zunächst in eine Gruppe von Teilgebieten, den Elementen, unterteilt wird. Wählt man zu wenige Elemente für das jeweilige Problem, wird die Lösung allerdings ungenau. Je mehr Elemente man jedoch nutzt, desto höher wird die Rechenzeit. Nimmt man hier die Gitterpunkte in einer Koordinatenrichtung als Parameter N, so müssen in einem dreidimensionalen Problem N^3 lokale Matrizen berechnet werden. Somit muss die Berechnung auf den einzelnen Elementen so effizient wie möglich ablaufen, um die Rechenzeit zu minimieren. Im Rahmen dieser Bachelorarbeit wurde deshalb untersucht, für spezifische Elementtypen ein möglichst effizientes Programm zur Berechnung der lokalen Steifigkeitsmatrizen der einzelnen Elemente zu schreiben und dessen Performanz mit dem aktuell in Kombination mit dem Finite-Elemente-Code von UGBlocks genutzten Programms Colsamm zu vergleichen. Zudem wurden weitere Ansätze zur Effizienzsteigerung, unabhängig von der Implementierung der lokalen Steifigkeitsmatrixberechung, untersucht.

3 Methode der Finiten Elemente

Wie bereits beschrieben, handelt es sich bei der Methode der Finiten Elemente um ein Verfahren zur Lösung von Differentialgleichungen. Der Prozess wird hier anhand des Beispiels der Poisson-Gleichung kurz erklärt. Diese lautet:

$$-\Delta u = f \tag{1}$$

Das Ziel ist, eine Annäherung der Funktion u auf einem Gebiet G zu finden. Dieses wird hierfür in Elemente unterteilt. Für ein zweidimensionales Gebiet können beispielsweise dreiecksförmige Elemente genutzt werden. Die Funktion u ist hier in einem mehrdimensionalen Gebiet definiert und in der Gleichung doppelt abgeleitet, dargestellt durch den Laplace-Operator. Um eine Annäherungslösung für u zu finden, wird auf beiden Seiten der Gleichung eine Funktion v anmultipliziert.

$$-\Delta uv = fv \tag{2}$$

Die Funktion v ist hierbei so gewählt, dass sie sowohl auf den einzelnen Elementen linear ist als auch am Rand des betrachteten Gebiets den Wert null hat. Nun werden beide Seiten der Gleichung über das gesamte Gebiet integriert:

$$\int_G -\Delta uv \, dG = \int_G fv \, dG \tag{3}$$

Auf der linken Seite der Gleichung wird nun partielle Integration angewendet. Da die Funktion v so gewählt wurde, dass sie auf dem Rand des betrachteten Gebiets den Wert null hat, bleibt folgende Gleichung:

$$\int_G \nabla u \nabla v \, dG = \int_G fv \, dG \tag{4}$$

Als letzter Schritt werden nun die Funktionen u und f durch Approximationen ersetzt, die wir als $\hat{u}$ und $\hat{f}$ bezeichnen. Diese sind genau wie v elementweise lineare Funktionen. Diese Funktionen werden nun als Summe einzelner Funktionen, nämlich exakt einer pro Gitterpunkt der Diskretisierung, die jeweils mit einem Gewicht multipliziert werden, betrachtet. Für v wird dasselbe getan, allerdings können durch den bei der Wahl von v gegebenen Spielraum die Gewichte so gewählt werden, dass sie alle den Wert eins haben. Dadurch fällt bei $\hat{v}$ die Multiplikation mit Gewichten weg. Die Teilfunktionen sind so gewählt, dass es für jeden Gitterpunkt genau eine gibt, die an diesem Gitterpunkt den Wert eins und an allen anderen den Wert null hat.

$$\int_G \nabla \hat{u} \nabla \hat{v} \, dG = \int_G \hat{f} \hat{v} \, dG \tag{5}$$

Zur Lösung dieser Gleichung soll nun nach $\hat{u}$ gelöst werden. Als elementweise lineare Funktionen, aus denen $\hat{u}$ gebildet wird, werden dieselben verwendet die auch $\hat{v}$ bilden. Somit sind die einzig verbleibenden Unbekannten die Gewichte. Für $\hat{f}$ wird analog vorgegangen. Nun kann die linke sowie die rechte Seite der Gleichung als Matrizen, multipliziert mit den Gewichtsvektoren, die wir als X beziehungsweise F bezeichnen, betrachtet werden. Die Matrix auf der linken Seite der Gleichung wird ab sofort als L-Matrix und die auf der rechten Seite als H-Matrix bezeichnet. Dies folgt der Bezeichung in den Programmbeispielen von UGBlocks. Somit entsteht die folgende Gleichung:

$$LX = HF \tag{6}$$

Die Einträge der Matrix L sind $l_{i,j} = \int_G \nabla \hat{u}_i \nabla \hat{v}_j \, dG$, wobei i und j die Indizes der Gridpunkte der Gebietsdiskretisierung sind. Analog sind die Einträge der Matrix H: $h_{i,j} = \int_G \hat{u}_i \hat{v}_j \, dG$. Die Werte des Gewichtsvektors F sind durch die Funktion f bekannt, während die Werte des Vektors X diejenigen sind, nach denen das Gleichungssystem nach Aufstellen der Matrizen L und H gelöst werden muss, um eine Lösung für $\hat{u}$ zu bekommen. Da das Gebiet G in Elemente aufgeteilt wurde, können wir das Integral über das gesamte Gebiet als Summe der Integrale über die einzelnen Elemente der Gebietsdiskretisierung sehen. N ist hierbei die Anzahl der Elemente, n die Indizes der einzelnen Elemente T_n:

$$\int_G \nabla \hat{u}_i \nabla \hat{v}_j \, dG = \sum_{n=1}^{N} \int_{T_n} \nabla \hat{u}_i \nabla \hat{v}_j \, dT_n \tag{7}$$

Da die Funktionen $\hat{u}$ und $\hat{v}$ elementweise linear sind und an allen Gitterpunkten außer einem den Wert null haben, ergeben die Summanden der obigen Summe nur dann einen Wert ungleich null, wenn die Gitterpunkte mit den Indizes i und j Eckpunkte des Dreiecks T_n sind. Da es im Fall von Dreieckselementen drei solche Indizes für i und j gibt, die frei miteinander kombiniert werden können, gibt es also für jedes Dreieckselement neun Einträge in der globalen Steifigkeitsmatrix, bei denen das Dreieck einen Anteil an der Berechnung dieser Einträge nach obiger Formel hat. Diese neun Eintragsanteile können also für jedes Element getrennt berechnet werden, und anschließend in der globalen Steifigkeitsmatrix als Summanden eingetragen werden. Die 3x3-Matrizen, die hierfür pro Element berechnet werden müssen, heißen lokale Steifigkeitsmatrizen. Sie sind folgendermaßen aufgebaut:

$$(\int_{T_n} \nabla \hat{u}_i \nabla \hat{v}_j \, dT_n)_{i,j \in T_n} \tag{8}$$

Die H-Matrix wird analog zusammengesetzt, die lokalen Matrizen zu ihrer Berechnung haben die Form:

$$(\int_{T_n} \hat{u}_i \hat{v}_j \, dT_n)_{i,j \in T_n} \tag{9}$$

4 Berechnung lokaler Steifigkeitsmatrizen auf Dreieckselementen

In diesem Kapitel wird der komplette Prozess der Berechnung lokaler Steifigkeitsmatrizen für die mehrdimensionale Poisson Gleichung für eine Dreieckselementdiskretisierung erklärt. In den folgenden Kapiteln wird auf die nötigen Anpassungen und Formeln für Vierecke, Tetraeder und Hexaeder eingegangen, wobei Hexaeder insbesondere im Hinblick auf die Effizienzsteigerung der Berechnung im Fokus liegen.

4.1 Referenzelement und Basisfunktionen

Um die einzelnen lokalen L- und H-Matrizen zu berechnen, müssen auf dem entsprechenden Element sämtliche Kombinationen der stückweise linearen Funktionen, aus denen $\hat{u}$ und $\hat{v}$ aufgebaut sind, miteinander multipliziert werden. Diese sind so gewählt, dass sie den sogenannten Basisfunktionen entsprechen. Für jedes Element ist die Anzahl dieser Basisfunktionen, die nicht auf dem gesamten Element gleich null sind, gleich der Anzahl seiner Eckpunkte. Diese sind so zu wählen, dass es pro Eckpunkt jeweils eine gibt, die den Wert eins annimmt, und an den anderen Eckpunkten den Wert null. Im Fall eines zweidimensionalen Gebiets ist eine der Möglichkeiten, dieses zu diskretisieren, die Unterteilung in Dreiecke. Während eine gute Diskretisierung zwar diverse Ansprüche an die Form dieser Dreiecke hat, wie zum Beispiel eine Mindestgröße ihrer Winkel, so können sich die Elemente dennoch stark unterscheiden im Bezug auf ihre Lage im zweidimensionalen Raum, sowie der genauen Größe ihrer Winkel. Das Aufstellen der Basisfunktionen müsste somit theoretisch für jedes einzelne Element durchgeführt werden und könnte je nach Form und Position des Dreiecks relativ aufwändig sein. Deshalb wird nicht auf den Elementen mit ihren tatsächlichen Koordinaten gerechnet, sondern eine Transformation auf ein Referenzelement durchgeführt:

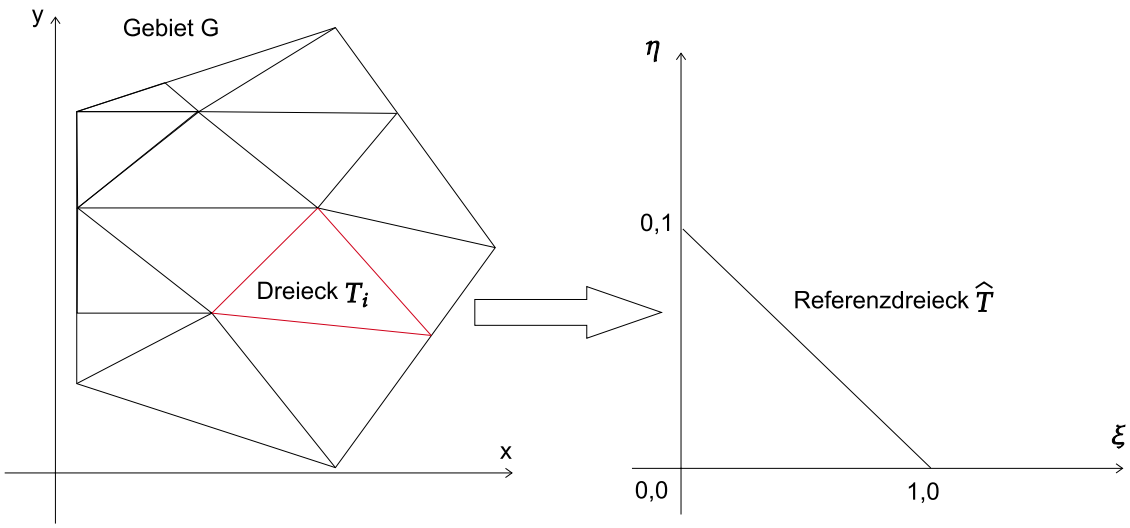

Abbildung 1: Transformation eines Elements der Diskretisierung auf das Referenzelement

Die Basisfunktionen sind auf dem Referenzdreieck leicht aufzustellen. Zu jedem Eckpunkt wird eine

zugehörige lineare Funktion benötigt die an diesem Punkt den Wert eins, und den anderen beiden den Wert null hat. Die Basisfunktionen werden hier u_i genannt, wobei i der Index des zugehörigen Punktes ist. In der Literatur beginnt der Index oft bei null, hier wurde zwecks besserer Nachvollziehbarkeit der Startindex eins gewählt. Die Basisfunktionen sind somit:

P1(0,0)	u_1	1 - $\xi - \eta$
P2(1,0)	u_2	ξ
P3(0,1)	u_3	η

Tabelle 1: Basisfunktionen Referenzdreieck

Die Achsen des Koordinatensystems des Referenzelements werden hierbei ξ und η genannt, da x und y bereits im originalen Koordinatensystem der tatsächlichen Elemente genutzt werden. Durch die Nutzung dieser immer gleichen und relativ simplen Basisfunktionen ist auch die Integration der jeweiligen Ausdrücke leichter. Allerdings muss hierfür noch die Koordinatentransformation für das zu berechnende Integral durchgeführt werden. Die Umformung wird durch eine mehrdimensionale Funktion vorgenommen, die wir hier mit Ψ bezeichnen. Ψ_x ist die Komponente zur Berechnung der zugehörigen x-Koordinate, und Ψ_y bildet auf die y-Koordinate ab. Zum Aufstellen dieser Funktionen werden die Eckpunkte als Fixpunkte genommen. Durch Multiplikation der den jeweiligen Eckpunkten zugeordneten Basisfunktionen mit den x beziehungsweise y Koordinaten selbiger Eckpunkte ist die korrekte Umrechnung sichergestellt:

$$\Psi_x = x_1 u_1(\xi, \eta) + x_2 u_2(\xi, \eta) + x_3 u_3(\xi, \eta) \tag{10}$$

$$\Psi_y = y_1 u_1(\xi, \eta) + y_2 u_2(\xi, \eta) + y_3 u_3(\xi, \eta) \tag{11}$$

Mithilfe dieser Transformationsfunktionen können nun die zu berechnenden Integrale für die ξ und η Koordinaten aufgestellt werden. Begonnen wird mit dem jeweiligen Integral im xy-Koordinatensystem.

4.2 Umformung des H-Matrix Integranden

Als erstes wird die Umformung der zur Berechnung der H-Matrix benötigten Integrale begonnen, da diese aufgrund der Abwesenheit von Gradienten simpler ist. Die zu integrierenden Ausdrücke haben hier die Form:

$$\int_T u_i(x, y) v_j(x, y)\, dT \tag{12}$$

Die Indizes i und j gehen jeweils von eins bis drei, und integriert wird in dieser Formel noch über das tatsächliche Element in x und y Koordinaten. Es fehlt also noch eine Transformation auf das Referenzelement. Verwendet wird hierbei der Transformationssatz für mehrdimensionale Integrale. Mit diesem erhält man folgenden Ausdruck:

$$\int_{\hat{T}} \hat{u}(\xi, \eta) \hat{v}(\xi, \eta) det(D\Psi)\, d\hat{T} \tag{13}$$

Der Faktor $det(D\Psi)$ ist hierbei die Determinante der Jakobimatrix, die der Transformation zugehörig ist. Die Jakobimatrix wird folgendermaßen gebildet:

$$D\Psi = \begin{pmatrix} \frac{\partial \Psi_x}{\partial \xi} & \frac{\partial \Psi_x}{\partial \eta} \\ \frac{\partial \Psi_y}{\partial \xi} & \frac{\partial \Psi_y}{\partial \eta} \end{pmatrix}$$

Da zur Berechnung der Determinanten lineare Ausdrücke, die nach ξ abgeleitet wurden, nur mit linearen Ausdrücken, die nach η abgeleitet wurden, multipliziert werden (und andersherum), ist ihr Wert unabhängig von ξ und η. Multipliziert mit den diversen Basisfunktionen enstehen somit Polynome, die maximal den Grad zwei haben. Dies ist später relevant für die Wahl der Formel zur numerischen Integration.

4.3 Umformung des L-Matrix Integranden

Die Einträge der L-Matrix entsprechen der Formel:

$$\int_T \nabla u(x,y) \nabla v(x,y)\, dT \tag{14}$$

Die Transformation der L-Matrix ist um einiges komplizierter, da der Gradient berücksichtigt werden muss. Die Funktion Ψ bildet von dem (ξ, η) - KOS auf das (x,y) - KOS ab. Somit gilt:

$$\hat{v}(\xi,\eta) = v \circ \Psi(\xi,\eta)) \tag{15}$$

Durch rechtsseitiges Verketten mit der Inversen Transformationsfunktion erhällt man:

$$\hat{v} \circ \Psi^{-1}(x,y) = v(x,y) \tag{16}$$

Setzt man dies nun in den Gradienten ein, so erhält man:

$$\nabla_{x,y}(\hat{v} \circ \Psi^{-1}(x,y)) = \nabla v(x,y) \tag{17}$$

Hierbei ist zu beacthen, dass der Gradient nach wie vor auf beiden Seiten nach x beziehungsweise y abgeleitet wird. Nun kann man den gesamten Ausdruck mittels des Transformationssatzes zum Referenzelement-KOS transformieren und erhält den Ausdruck:

$$\int_T \nabla v(x,y)\, dT = \int_{\hat{T}} \nabla_{x,y}(\hat{v} \circ \Psi^{-1}(x,y)) \circ \Psi(\xi,\eta) det(D\Psi)\, d\hat{T} \tag{18}$$

Durch Anwendung der mehrdimensionalen Kettenregel und Kürzen der hierbei entstehenden Multiplikation mit der Einheitsmatrix erhält man aus Gleichung (18):

$$\int_T \nabla v(x,y)\, dT = \int_{\hat{T}} (\nabla_{\xi,\eta} \hat{v}(\Psi^{-1}) \circ \Psi(\xi,\eta)) det(D\Psi)\, d\hat{T} \tag{19}$$

An dieser Stelle wird eine weitere Gleichung benötigt, die im folgenden hergeleitet wird:
$\Psi^{-1} \circ \Psi$ angewendet auf einen Vektor, der als Einträge die ξ und η hat, ergibt nach Definition des inversen einer mehrdimensionalen Funktion wieder den (ξ,η)-Vektor. Differenziert man nun auf beiden Seiten, erhält man:

$$D(\Psi^{-1} \circ \Psi) = E \tag{20}$$

E steht hierbei für die Einheitsmatrix. Nun wird erneut die mehrdimensionale Kettenregel angewandt und man erhält:.

$$D(\Psi^{-1} \circ \Psi) D\Psi = E \tag{21}$$

Multipliziert man die Gleichung nun rechtsseitig mit $(D\Psi)^{-1}$, erhält man:

$$D(\Psi^{-1} \circ \Psi) = D(\Psi)^{-1} \tag{22}$$

Dies lässt sich nun in Gleichung 17 einsetzen, es ensteht folgende Gleichung:

$$\int_T \nabla v(x,y)\, dT = \int_{\hat{T}} (\nabla_{\xi,\eta} \hat{v} D(\Psi)^{-1}) det(D\Psi)\, d\hat{T} \tag{23}$$

Die Reihenfolge der Matrix-Vektor-Multiplikation von $\nabla_{\xi,\eta}\hat{v}$ und $D(\Psi)^{-1}$ lässt sich ohne Verlust der Äquivalenz umkehren, solange man gleichzeitig auch transponiert. Außerdem lässt sich die komplette Transformation von Gleichung 13 bis Gleichung 21 auch für $\nabla_{\xi,\eta}\hat{u}$ durchführen. Setzt man den umgeformten Term nun also zweimal in Gleichung 12 ein, erhält man den Ausdruck, der im Weiteren tatsächlich zur Berechnung der L-Matrix verwendet wird:

$$\int_{\hat{T}} ((D\Psi)^{-T} \nabla_{\xi,\eta}\hat{u})^T ((D\Psi)^{-T} \nabla_{\xi,\eta}\hat{v}) det(D\Psi)\, d\hat{T} \tag{24}$$

4.4 Gauss-Legendre Quadratur

Nachdem die auszuwertenden Integrale nun aufgestellt sind, müssen diese noch berechnet werden. Hierbei wird die Gauss-Legendre Quadratur genutzt, bei welcher lediglich die Werte der zu integrierenden Funktion an bestimmten Punkten des Elements berechnet und mit vorher berechneten Gewichten w_i multipliziert werden müssen:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i)w_i \tag{25}$$

Es ist bewiesen, dass mit n korrekt gewählten Quadraturpunkten (in jeder Dimension) Polynome des Grads 2n-1 exakt integriert werden können. Da die Gradienten linearer Basisfunktionen konstant sind, reicht ein Quadraturpunkt zur Berechnung der Einträge der L-Matrix, da diese von Grad null sind. Im Fall der H-Matrix werden lineare Funktionen innerhalb des Integrals multipliziert, somit entstehen durch die Multiplikation mit der Determinanten, die unabhängig von ξ und ηist,maximal Polynome des Grades zwei. Daher müssen in jede Richtung zwei Punkte genutzt werden. Allerdings ist es im Dreiecksfall hinreichend, insgesamt drei Punkte zu nutzen. Die jeweiligen Punkte und Gewichte finden sich in der folgenden Tabelle.

Stützpunkte	ξ	η	Gewicht
1	1/3	1/3	1
3	1/6	1/6	1/3
	2/3	1/6	1/3
	1/6	2/3	1/3

Tabelle 2: Gauss-Legendre-Punkte Referenzdreieck

Zusätzlich muss am Ende einer Gauss-Quadratur noch die Fläche beziehungsweise das Volumen des Gebiets, über das integriert wurde, mit dem Ergebnis multipliziert werden, im Fall des Referenzdreiecks fehlt also noch eine Multiplikation mit dem Faktor 0.5.

4.5 Lokale Steifigkeitsmatrix

Zur Berechnung der einzelnen Einträge der lokalen Steifigkeitsmatrizen müssen jetzt die Integrale die in den Formeln (13) und (24) gegeben sind mittels der Gauss-Legendre Quadratur berechnet werden. Hierfür wurde ein Programm geschrieben, dem die Koordinaten der Eckpunkte des zu berechnenden Elements übergeben werden. Zusätzlich kann mittels verschiedener Parameter angegeben werden um welchen Elementtyp es sich handelt, wie viele Gausspunkte genutzt werden sollen, und ob eine H-Matrix oder eine L-Matrix zurückgegeben werden soll. Die hierfür verwendete Methode ruft pro benötigtem Gausspunkt eine weitere Methode zur Berechnung des Anteils dieses Gausspunktes an den einzelnen Einträgen der lokalen Steifigkeitsmatrix auf. Hierfür werden dieser Funktion die Koordinaten des zugehörigen Gausspunktes auf dem Referenzelement, sowie die tatsächlichen Koordinaten des Elements aus der Gebietsdiskretisierung übergeben. Mithilfe dieser

Werte wird zunächst die Jakobimatrix berechnet. Anschließend wird diese im Fall der Berechnung einer L-Matrix invertiert und transponiert, und später mittels Matrix-Vektor Multiplikation mit den Gradienten multipliziert. Dies wird für sämtliche zu berechnenden Einträge der lokalen Steifigkeitsmatrix, also im Dreiecksfall für alle drei mal drei Kombinationsmöglichkeiten der Basisfunktionen, durchgeführt. Zurückgegeben wird eine Matrix, die sämtliche Einträge des zum Gausspunkt gehörigen Anteils der lokalen Steifigkeitsmatrix beinhaltet. In der aufrufenden Methode werden diese Matrizen zusammenaddiert, wobei sie auch direkt mit den den jeweiligen Gausspunkten zugehörigen Gewichten mulitpliziert werden. Das Resultat ist die tatsächliche lokale Steifigkeitsmatrix des übergebenen Elements.

5 Berechnung lokaler Steifigkeitsmatrizen auf Vierecken, Tetraedern und Hexaedern

5.1 Zweidimensionale Viereckselemente

Die Berechnung auf viereckigen Elementen ist vom generellen Programmablauf ähnlich zu der Berechnung auf Dreiecken. Für zweidimensionale Viereckselemente wird in der im Rahmen dieser Bachelorarbeit angefertigten Implementierung ein Referenzelement genutzt, das auf beiden Koordinatenachsen von 0 bis 1 geht. In vielen Implementierungen sowie Beschreibungen in der Literatur werden Referenzelemente genutzt, die auf beiden Achsen von -1 bis 1 reichen. Der Grund für die Wahl der ersteren Variante ist, dass einige der zugehörigen Basisfunktionen für das [0,1] Element simpler sind. Was die Korrektheit der Umformung angeht, existiert kein Unterschied, allerdings muss die unterschiedliche Form im Bezug auf die Wahl der Gausspunkte und deren Gewichte berücksichtigt werden.

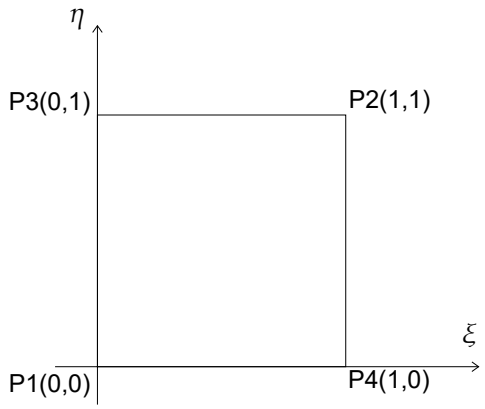

Abbildung 2: Referenzelement Viereck

Die den Punkten zugehörigen Basisfunktionen sind wie folgt:

P1(0,0)	u1	$(1-\xi)(1-\eta)$
P2(1,1)	u3	$\xi\eta$
P3(0,1)	u4	$(1-\xi)\eta$
P4(1,0)	u2	$\xi(1-\eta)$

Tabelle 3: Basisfunktionen Referenzviereck

Die Transformationsfunktion bildet sich exakt wie bei den Dreieckselementen, allerdings müssen jetzt vier Punkte und vier Basisfunktionen genutzt werden.

$\Psi_x = x_1 u_1(\xi, \eta) + x_2 u_2(\xi, \eta) + x_3 u_3(\xi, \eta) + x_4 u_4(\xi, \eta)$
$\Psi_y = y_1 u_1(\xi, \eta) + y_2 u_2(\xi, \eta) + y_3 u_3(\xi, \eta) + y_4 u_4(\xi, \eta)$

Auch die Integrale zur Berechnung der H-Matrix und der L-Matrix bleiben gleich, abgesehen davon dass nun über das Referenzviereck integriert wird. Die Gaussintegration benötigt jetzt, je nach Polynom , einen beziehungsweise zwei Stützpunkte in jede Richtung. Der zu integrierende Eintrag der H-Matrix ist vom Grad 3, somit werden hier 2^2 Punkte benötigt. Für die L-Matrix ist das Bestimmen der Gausspunkte etwas schwieriger, da die Basisfunktionen bilinear sind und somit im Inversen der Jakobimatrix ξ und η in den Nennern der Einträge vorkommt. In Kapitel 5.3 wird allerdings experimentell für Hexaeder gezeigt, dass zwei Gausspunkte pro Koordinatenachse hinreichend sind, was sich auch für Vierecke generalisieren lässt. Die jeweiligen Punkte finden sich in der nachfolgenden Tabelle. Hier ist zu berücksichtigen, dass viele Tabellen mit Integrationspunkten in der Literatur von dem Intervall [-1,1] ausgehen. Um diese Werte auf das hier genutzt Referenzelement anzupassen, muss die Umformung $(\xi + 1)/2$ beziehungsweise $(\eta + 1)/2$ für die Integrationspunkte vorgenommen werden.

Stützpunkte	ξ	η	Gewicht
1	0.5	0.5	1
4	0.2113248654	0.2113248654	1/4
	0.78867513459	0.2113248654	1/4
	0.78867513459	0.78867513459	1/4
	0.2113248654	0.78867513459	1/4

Tabelle 4: Gauss-Legendre-Punkte Referenzviereck

Das Programm zur Berechnung der H-Matrix beziehungsweise der L-Matrix folgt, abgesehen von den Unterschieden in den Basisfunktionen, und somit auch ihren Ableitungen in der Jakobimatrix, sowie den unterschiedlichen Gewichten und Gausspunkten, dem selben Ablauf wie bereits bei den Dreieckselementen.

5.2 Tetraeder Elemente

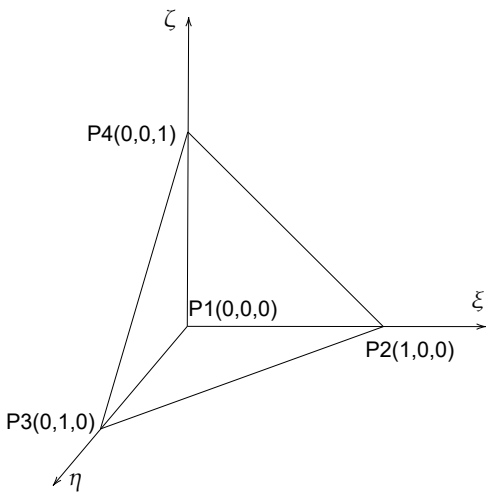

Abbildung 3: Referenzelement Tetraeder

Da Tetraeder Körper im dreidimensionalen Raum sind, benötigen wir auch eine dreidimensionale Transformationsfunktion. Die einzelnen Funktionen folgen jedoch nach wie vor dem Muster, das wir bei den Dreiecks-Elementen aufgestellt haben. Die Basisfunktionen des Referenztetraeders sind:

P1(0,0)	u1	$1 - \xi - \eta - \zeta$
P2(1,0)	u2	ξ
P3(1,1)	u3	η
P4(0,1)	u4	ζ

Tabelle 5: Basisfunktionen Tetraeder

Die Transformationsfunktion ist jetzt eine dreidimensionale Funktion, das heisst sie gibt einen x, y und einen z Wert zurück. Die einzelnen Anteile sehen aus wie folgt:

$$\Psi_x = x_1 u_1(\xi,\eta,\zeta) + x_2 u_2(\xi,\eta,\zeta) + x_3 u_3(\xi,\eta,\zeta) + x_4 u_4(\xi,\eta,\zeta)$$
$$\Psi_y = y_1 u_1(\xi,\eta,\zeta) + y_2 u_2(\xi,\eta,\zeta) + y_3 u_3(\xi,\eta,\zeta) + y_4 u_4(\xi,\eta,\zeta)$$
$$\Psi_z = z_1 u_1(\xi,\eta,\zeta) + z_2 u_2(\xi,\eta,\zeta) + z_3 u_3(\xi,\eta,\zeta) + z_4 u_4(\xi,\eta,\zeta)$$

Was die Integrationspunkte angeht, so sind diese im Vergleich zum Dreieck etwas verschoben. Dies liegt daran, dass der Schwerpunkt des Referenztetraeders im Vergleich zu dem Referenzdreieck auf den einzelnen Achsen näher am Koordinatenursprung liegt. Die Zahl der Gausspunkte pro Koordinatenrichtung ist jedoch sowohl für die H-Matrix als auch die L-Matrix exakt wie bei den

Dreieckselementen zu wählen, da sich an dem Grad der Polynome in den Integralen nichts geändert hat.

Stützpunkte	ξ	η	ζ	Gewicht
1	0.25	0.25	0.25	1
4	0.13819660	0.13819660	0.13819660	1/4
	0.5841020	0.13819660	0.13819660	1/4
	0.13819660	0.5841020	0.13819660	1/4
	0.13819660	0.13819660	0.5841020	1/4

Tabelle 6: Gauss-Legendre-Punkte Referenztetraeder [Rei19]

Hier muss das Ergebnis am Ende noch mit dem Volumen des Referenztetraeders multipliziert werden, also mit dem Faktor 1/6.

Während die grundlegende Reihenfolge der Schritte zur Berechnung der L-Matrix sowie der H-Matrix auch für den Tetraeder gleich bleibt, unterscheidet sich die Implementierung und die Laufzeit aufgrund der Dimension des betrachteten Gebiets. Für eine Transformation dreidimensionaler Elemente ist die Jakobimatrix eine 3x3-Matrix. Dadurch mussten neue Methoden zum Berechnen der Determinanten, sowie dem Transponieren und Invertieren der Matrix implementiert werden. Auch sind die Gradienten in den Integranden der L-Matrix jetzt Vektoren der Länge drei.

5.3 Hexaeder Elemente

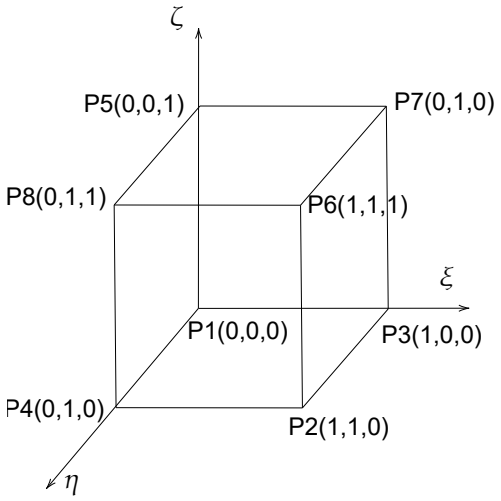

Abbildung 4: Referenzelement Hexaeder

Da Hexaeder acht Eckpunkte haben, werden insgesamt 8 Basisfunktionen benötigt. Der Aufbau dieser ist ähnlich wie bereits bei den zweidimensionalen Viereckselementen.

P1(0,0)	u1	$(1-\xi)(1-\eta)(1-\zeta)$
P2(1,0)	u2	$\xi(1-\eta)(1-\zeta)$
P3(1,1)	u3	$\xi\eta(1-\zeta)$
P4(0,1)	u4	$(1-\xi)\eta(1-\zeta)$
P1(0,0)	u5	$(1-\xi)(1-\eta)\zeta$
P2(1,0)	u6	$\xi(1-\eta)\zeta$
P3(1,1)	u7	$\xi\eta\zeta$
P4(0,1)	u8	$(1-\xi)\eta\zeta$

Tabelle 7: Basisfunktionen Referenzhexaeder

Die Funktionen sind jetzt trilinear, dass heißt die ξ, η und ζ Komponenten werden innerhalb der einzelnen Basisfunktionen miteinander multipliziert, was später zu Ausdrücken höherer Ordnung in den Gradienten und Jacobimatrizen führt. Das Referenzelement erstreckt sich, analog zum viereckigen Referenzelement, auf allen Koordinatenachsen von null bis eins.
Die Gausspunkte für die Integration über den Referenzhexaeder sind ebenfalls in ihren Werten ähnlich zu denen des Vierecks, allerdings müssen jetzt x^3 Punkte genutzt werden. Aufgrund der gebrochen rationalen Faktoren innerhalb des Integranden durch das Inverse der Jakobimatrix ist für die Berechnung der L-Matrix nicht bewiesen, dass zwei Gausspunkte in jeder Koordinatenrichtung ausreichen, um das Integral exakt zu berechnen. Aus diesem Grund wurde ebenfalls eine

Integration mit drei hoch drei Gausspunkten implementiert, um zu testen, ob diese zu einer exakteren Integration der lokalen Steifigkeitsmatrizen und somit eventuell zu höherer Genauigkeit für die Lösung der Poisson-Gleichung beziehungsweise besserer Konvergenz des LGS-Lösers führt. Die Gausspunkte der verschiedenen Genauigkeiten finden sich in der nachfolgenden Tabelle. Im Fall der Nutzung von 3^3 Punkten werden sämtliche Kombinationen der Werte $a1 = 0.11270166538$, $a2 = 0.5$ und $a3 = 0.88729833462$ als Koordinaten genutzt. Auch sind die Gewichte nicht mehr für jeden Punkt gleich, sondern höher für Punkte, die näher am Schwerpunkt des Element liegen. Zur Berechnung der Gewichte wird für jede Koordinate ein Teilgewicht genommen, und diese drei Teilgewichte miteinander multipliziert. Da a1 und a3 die gleiche Distanz zur Mitte des [0,1] Intervalls haben, haben sie das gleiche Teilgewicht. Dieses beträgt $w_{side} = 0.2777777778$. Das Teilgewicht, das zu a2 gehört, nennen wir hier $w_{central}$ und beträgt 0.4444444444. Zum Beispiel würde der Gausspunkt an den Koordinaten (0.11270166538, 0.5, 0.88729833462) also mit dem Gewicht $w_{side}w_{central}w_{side} = 0.03429355281$ gewichtet werden.

Stützpunkte	ξ	η	ζ	Gewicht
1	0.5	0.5	0.5	1
8	0.2113248654	0.2113248654	0.2113248654	1/8
	0.2113248654	0.2113248654	0.78867513459	1/8
	0.2113248654	0.78867513459	0.2113248654	1/8
	0.2113248654	0.78867513459	0.78867513459	1/8
	0.78867513459	0.2113248654	0.2113248654	1/8
	0.78867513459	0.2113248654	0.78867513459	1/8
	0.78867513459	0.78867513459	0.2113248654	1/8
	0.78867513459	0.78867513459	0.78867513459	1/8
27	0.11270166538	0.11270166538	0.11270166538	0.0214334705
	0.11270166538	0.11270166538	0.5	0.03429355281
	.....	.....	.....	.....
	.....	.....	.....	.....
	0.88729833462	0.88729833462	0.5	0.03429355281
	0.88729833462	0.88729833462	0.88729833462	0.0214334705

Tabelle 8: Gauss-Legendre-Punkte Hexaeder

Um zu überprüfen, ob die Nutzung von insgesamt 27 Gausspunkten tatsächlich bessere Matrizen erzeugt, wurde das Programm so modifiziert, dass es die Koordinaten einzelner Elemente aus der durch UGBlocks erzeugten Diskretisierung ausliest, die Matrizen für beide Genauigkeiten berechnet und sowohl die Matrizen, als auch eine Matrix, deren Einträge aus den Differenzen der Einträge der beiden Matrizen besteht, ausgibt. Hier konnten auf mindesten zehn Nachkommastellen keine Unterschiede in den berechneten Matrizen festgestellt werden. Auch ergab ein Lösen der Poisson-Gleichung auf einem Zylinder unter Nutzung der beiden verschiedenen Implementierungen keine Unterschiede bezüglich der Konvergenzgeschwindigkeit. Somit ist davon auszugehen, dass eine Nutzung von 27 Gausspunkten keinen Vorteil bietet. Auch in der Colsamm-Implementierung werden standardmäßig für das Berechnen der L-Matrix auf Hexaedern acht Gausspunkte genutzt.

6 Optimierung der Berechnung lokaler Steifigkeitsmatrizen

In diesem Kapitel werden verschiedene Programme und Programmvariationen zur Berechnung lokaler Steifigkeitsmatrizen auf ihre Performanz geprüft. Die Spezifikationen des Systems, auf dem sämtliche Laufzeitmessungen durchgeführt wurden, sind wie folgt:

Betriebssystem:	Kubuntu 20.04 (64-bit)
Prozessoren:	8 × Intel® Xeon® CPU E3-1275 v5 @ 3.60GHz
Arbeitsspeicher:	62.6 GiB RAM

Die Programme wurden für die Tests in das Programm UGBlocks eingebunden, welches genutzt wurde, um auf zylindrischen Körpern Hexaeder-Grids mit variablen Gridpunktzahlen zu generieren. Sämtliche Tests wurden für eine Diskretisierung mit 10 Punkten pro Dimension des Körpers, mit 30 Punkten pro Dimension, sowie mit 50 Punkten pro Dimension durchgeführt. Gemessen wurde die Dauer der Berechnung der L-Matrix, die Dauer der Berechnung der H-Matrix und die Gesamtdauer beider Berechnungen. Hierfür wurden Objekte einer Timer-Klasse benutzt, die ein Objekt der Klasse „high_ resolution_ clock“ aus der chrono-library verwendet. Zum Erzeugungszeitpunkt der Objekte wird der jeweilige objektinterne Timer gestartet, welcher dann mittels der „elapsed()“-Funktion jederzeit ausgelesen werden kann. Um sicherzustellen, dass die gemessenen Zeiten sinnvoll miteinander verglichen werden können, wurden sämtliche Messungen zehn mal durchgeführt und der Durchschnitt berechnet. In den seltenen Fällen, in denen eine Messung stark von den übrigen abwich, wurde diese wiederholt. Zudem wurde zu Beginn jeder späteren Testreihe ein Aufruf der Implementierung mittels Colsamm für 30 Gridpunkte ausgeführt und die Laufzeit mit früheren Ergebnissen verglichen um sicherzustellen, dass sich an der effektiven Leistungsfähigkeit des Systems nichts geändert hat, zum Beispiel durch etwaige Hintergrundprozesse.

6.1 Performanz der nicht optimierten Implementierung

Um einen Vergleichswert für die einzelnen Optimierungen, die im Rahmen der versuchten Effizienzsteigerung umgesetzt wurden, zu haben, wurde zunächst die Laufzeit der unoptimierten Erstimplementierung T_{notOpt} des eigenen Programms gemessen. Zusätzlich wurde die Laufzeit unter Nutzung von Colsamm T_{Col} gemessen, und mit der eigenen Implementierung verglichen. In der nachfolgenden Tabelle finden sich die Laufzeiten sowie die prozentuale Verbesserung durch Colsamm.

Gitterpunkte	Matrizen	T_{notOpt}	T_{Col}	$1 - T_{Col}/T_{notOpt}$
10	L-Matrix	0.1937482	0.10955	-43.45753%
	H-Matrix	0.08712882	0.0783575	-10.06707%
	Beide	0.2809168	0.187949	-33.09442%
30	L-Matrix	1.718218	0.9298768	-45.88132%
	H-Matrix	0.7955212	0.687959	-13.52097%
	Beide	2.513786	1.617882	-35.63962%
50	L-Matrix	4.808702	2.631772	-45.27063%
	H-Matrix	2.247628	1.971842	-12.27009%
	Beide	7.05638	4.603664	-34.75884%

Tabelle 9: Vergleich Laufzeit der eigenen Implementierung zu Colsamm

Es ist zu sehen, dass Colsamm für alle getesteten Problemgrößen circa 43 bis 46 Prozent schneller

in der Berechnung der L-Matrix ist als die eigene Erstimplementierung. Für die H-Matrix liegt der Speedup lediglich bei 10 bis 15 Prozent. Insgesamt ergibt sich somit für die Berechnung beider Matrizen aufgrund der höheren Gesamtdauer der Berechnung der L-Matrix ein Speedup von 33 bis 36 Prozent. Die Unterschiede der Berechnungsdauer für die beiden Matrizen sind dadurch zu erklären, dass sämtliche Matrixoperationen, wie das Invertieren und Transponieren, sowie die zahlreichen Vektor-Matrix Multiplikationen sowie Vektor-Vektor Multiplikationen, die zur Berechnung der Einträge der L-Matrix notwendig sind, bei der H-Matrix entweder gänzlich wegfallen oder durch Multipikation von Skalaren umgesetzt werden. Durch die insgesamt geringere Berechnungskomplexität ist auch weniger Potential für Optimierungen gegeben, und die in der Laufzeit nicht verbesserbaren Anteile, wie das Auslesen der Elementeckpunkte aus der UGBlocks-Diskretisierung, fallen prozentual mehr ins Gewicht. Dies erklärt den geringeren Speedup durch die effizientere Colsamm Implementierung bei der Berechnung der H-Matrix im Vergleich zur L-Matrix.

6.2 Einsparen doppelter Berechnungen

Wenn man sich die Berechnung der Einträge der beiden Matrizen anschaut, so ist zu sehen, dass viele Ausdrücke mehrfach benötigt werden. Somit kann es sinnvoll sein, diese einmal zu berechnen und zwischenzuspeichern, um Rechenzeit zu sparen. Bei der Berechnung der H-Matrix werden beispielsweise sämtliche acht Basisfunktionen in allen möglichen Kombinationen miteinander multipliziert. Somit ist es möglich, bei jeder Berechnung für einen Gausspunkt diese acht Ausdrücke auszurechnen und zwischenzuspeichern, sodass sie nicht 16-mal berechnet werden müssen. Hier werden allerdings lediglich Additionen und Subtraktionen eingespart. Im Fall der L-Matrix ist das Prinzip dasselbe, die Ausdrücke $((D\Psi)^{-T}\nabla_{\xi,\eta}\hat{u})$ können auch hier einmalig berechnet und zwischengespeichert werden. In diesem Fall spart man sich ebenfalls 15 weitere Auswertungen desselben Ausdrucks, und bei jeder eingesparten Auswertung wird hier eine Matrix-Vektor-Multiplikation sowie das Berechnen der einzelnen Einträge des Gradienten für die Koordinaten des aktuellen Gausspunktes eingespart. Die zweite Stelle im Programm, an der man sich unnötige Berechnungen sparen kann, ist beim Berechnen der endgültigen Matrixeinträge, beziehungsweise der Anteile des jeweiligen Gausspunktes an den Matrixeinträgen. Sowohl die H-Matrix als auch die L-Matrix ist symmetrisch entlang ihrer Diagonalen. Dies kommt daher, dass die jeweiligen Integranden denselben Wert ergeben, egal ob man Basisfunktion i zur Berechnung der ersten Hälfte des Ausdrucks und Basisfunktion j zur Berechnung der zweiten Hälfte verwendet, oder andersherum. Somit kann man durch Abrufen bereits berechneter Matrixeinträge die Menge nötiger Multiplikationen von 64 auf 46 senken. Im Fall der H-Matrix spart man sich somit 38 Mutlipikationen von Variablen des Typs double, im Fall der L-Matrix sogar 38 Vektor-Vektor-Multipikationen. In Tabelle 10 ist der Unterschied zwischen der Laufzeit der unoptimierten eigenen Implementierung T_{notOpt} zu der Laufzeit der Version, bei der oben genannte Berechnungen eingespart werden T_{red}, zu sehen:

Wie man erkennen kann, liefert die Optimierung im Fall der H-Matrix nur eine geringe Performanzverbesserung von ein bis drei Prozent. Dies liegt daran, dass die Berechnung der H-Matrix ohnehin nicht sonderlich aufwändig ist und andere Programmteile deutlich mehr ins Gewicht fallen als die hier eingesparten unnötigen Berechnungen. Im Fall der L-Matrix hingegen, bei der 120 Matrix-Vektor-Multiplikationen sowie 38 Vektor-Vektor-Multiplikationen eingespart wurden, lässt sich eine deutliche Verbesserung in der Rechenzeit von 12 bis 13 Prozent erkennen. Diese Ergebnisse sind über alle drei genutzten Gitterpunktzahlen zu beobachten. Aufgrund des eindeutigen positiven Effekts dieser Anpassung des Programms wird diese in allen weiteren Programmvariationen

ebenfalls genutzt.

Gitterpunkte	Matrizen	T_{notOpt}	T_{red}	$T_{red}/T_{notOpt} - 1$
10	L-Matrix	0.1937482	0.1599874	-17.42509%
	H-Matrix	0.08712882	0.08616282	-1.108703%
	Beide	0.2809168	0.2461908	-12.36167%
30	L-Matrix	1.718218	1.41425	-17.69089%
	H-Matrix	0.7955212	0.7725324	-2.889778%
	Beide	2.513786	2.186828	-13.00659%
50	L-Matrix	4.808702	3.946452	-17.93103%
	H-Matrix	2.247628	2.195368	-2.325118%
	Beide	7.05638	6.141868	-12.96007%

Tabelle 10: Laufzeitänderung durch Einsparen doppelter Berechnungen

6.3 Parallelisierung mit OpenMP

Im Rahmen der Suche nach einer möglichst effizienten Implementierung zur Berechnung der lokalen Steifigkeitsmatrizen wurde auch die Möglichkeit einer Parallelisierung mittels OpenMP [Boa21] untersucht. OpenMP ist eine API, die eine simple und effektive Parallelisierung von parallelisierbaren Programmen, unter anderem in C++, erlaubt. Ein möglicher Ansatzpunkt für eine Parallelisierung bietet sich bei der Berechnung der Anteile der einzelnen Gausspunkte an der fertigen Matrix. Da die Berechnung dieser Anteile unabhängig voneinander stattfindet und, insbesondere im Fall der L-Matrix, den Großteil der Programmlaufzeit ausmacht, können die acht Anteilsmatrizen bei Nutzung von 2^3 Gausspunkten gut parallel berechnet werden. Dennoch ist dies im betrachteten Anwendungsfall keine sinnvolle Maßnahme. Dies liegt daran, dass UGBlocks bereits eine OMP-Parallelisierungsoption anbietet, die durch das Setzen der booleschen Variable „useOpenMP“ auf „true“ aktiviert werden kann. Wird dies getan, so werden zahlreiche Stellen des UGBlocks-Codes parallelisiert, unter anderem die äußerste for-Schleife des Codeabschnitts, welcher für die Berechnung sämtlicher lokaler Steifigkeitsmatrizen verantwortlich ist. Somit tritt der Zusatzaufwand, der durch das Erstellen und Beenden der Threads generiert wird, lediglich einmal auf. Im Fall einer Parallelisierung innerhalb der Berechnung jeder einzelnen lokalen Steifigkeitsmatrix würde eben genannter Zusatzaufwand für jede Matrix anfallen, zum Beispiel bei einer Diskretisierung mit 1000 Elementen 1000-mal. Bereits aus diesem Grund ist durch eine interne Parallelisierung kein Vorteil gegenüber der bereits in UGBlocks implementierten Parallelisierung zu gewinnen. Hinzu kommt noch die Tatsache, dass es im betrachteten Anwendungsfall wesentlich mehr lokale Steifigkeitsmatrizen zu berechnen gibt, als Gausspunkte pro Matrix genutzt werden. Somit hat die UGBlocks-Parallelisierung auch ein höheres Limit für die Anzahl gleichzeitig sinnvoll nutzbarer Threads.

6.4 Vorausberechnen einzelner Faktoren der Integrale

In den zu integrierenden Ausdrücken der L-Matrix sowie der H-Matrix finden sich mehrere Teilausdrücke, die lediglich von den ξ, η und ζ Koordinaten abhängen, und unabhängig von den x, y und z Koordinaten des tatsächlichen, untransformierten Elements sind. Diese Teile der Formel sind somit identisch für die Berechnung jeder lokalen Steifigkeitsmatrix und müssen somit nicht in jedem Durchlauf des Programms neu berechnet werden. Im Fall der H-Matrix handelt es sich

um den gesamten Integrand mit Ausnahme der Determinante der Jakobimatrix. Im Fall der L-Matrix handelt es sich lediglich um die Gradienten, bevor sie mit $(D\Psi)^{-T}$ multipliziert werden. Im Versuch die Laufzeit der Matrixberechnungen weiter zu beschleunigen wurde deshalb der Ansatz verfolgt, diese Ausdrücke selbst zu berechnen und als feste Werte im Programm zu speichern, sodass sie lediglich ausgelesen und nicht neu berechnet werden müssen. Zu diesem Zweck wurde ein C++ Programm geschrieben, welches oben genannte Ausdrücke für die Koordinaten sämtlicher genutzter Gausspunkte berechnet und ausgibt. Die Ergebnisse wurden anschliesend in Vektoren gespeichert. Insgesamt mussten für die H-Matrix unter Verwendung von 2^3 Gausspunkten 36 Einträge pro Gausspunkt gespeichert werden, also insgesamt 288. Für die L-Matrix wurden 8 mal 24 Einträge benötigt, also 192. In der nachfolgenden Tabelle lässt sich sehen, wie sich die Laufzeit der Variante, bei der oben gelistete Ausdrücke zur Laufzeit berechnet werden (T_{rt}), von der Laufzeit der Anpassung, bei der die Ausdrücke in globalen Variablen gespeichert und den Funktionen als Parameter übergeben werden (T_{svd}), unterscheidet:

Gitterpunkte	Matrizen	T_{rt}	T_{svd}	$T_{svd}/T_{rt} - 1$
10	L-Matrix	0.1599874	0.168817	+5.518935%
	H-Matrix	0.08616282	0.0880538	+2.194659%
	Beide	0.2461908	0.2569122	+4.3549149%
30	L-Matrix	1.41425	1.465896	+3.651829%
	H-Matrix	0.7725324	0.7977364	+3.262517%
	Beide	2.186828	2.263678	+3.514222%
50	L-Matrix	3.946452	4.117886	+4.3440032%
	H-Matrix	2.195368	2.258326	+2.867765%
	Beide	6.141868	6.366262	+3.653514%

Tabelle 11: Laufzeitänderung durch Vorausberechnen von Faktoren

Wie zu erkennen ist, hat diese Anpassung in allen Szenarien, unabhängig von Gitterpunktzahl oder Matrixtyp, zu einer Laufzeitverschlechterung geführt. Das bedeutet, dass die Dauer der zusätzlichen Speicheraufrufe und der zusätzlichen Parameterübergaben den durch das Einsparen von Berechnungen gewonnenen Nutzen übertrifft. Grund hierfür dürfte sein, dass die eingesparten Berechnungen nicht sonderlich laufzeitintensiv sind. Im Fall der H-Matrix werden pro Gausspunkt lediglich eine Multiplikation und acht Summen zur Berechnung der acht Basisfunktionen auf den jeweiligen Koordinaten eingespart, sowie 44 Multiplikationen zweier double-Variablen zur Berechnung der einzigartigen Einträge der lokalen Steifigkeitsmatrix. Auf der Gegenseite stehen die zusätzlichen Speicheraufrufe und Parameterübergaben der vorberechneten Ausdrücke. Im Fall der L-Matrix wird ebenfalls nur wenig Rechendauer gespart, nämlich das Berechnen der Gradienten pro Gausspunkt, da die wesentlich aufwändigeren Matrix-Vektor in Abhängigkeit der tatsächlichen Elementkoordinaten stehen und ihre Ergebnisse somit erst zur Laufzeit berechnet werden können. Die Messungen zeigen, dass auch bei der L-Matrix die zusätzlichen Speicheraufrufe und Parameterübergaben den ersparten Rechenaufwand überwiegen und somit eine Laufzeitverschlechterung bewirken.

6.5 Linearisierung des Codes

In diesem Kapitel wird der Effekt der Linearisierung des Programms untersucht. Damit sind drei Schritte gemeint. Erstens wurden sämtliche If-Anweisungen aus dem Programm entfernt. Dazu

wurde die Implementierung grundsätzlich geändert, sodass dem Programm nicht mehr über Parameter mitgeteilt wird, welcher der beiden Matrixtypen berechnet werden soll, welcher Elementtyp durch die übergebenen Koordinaten beschrieben wird, und wie viele Gausspunkte verwendet werden sollen. Stattdessen wurden eigene Funktionen für jeden dieser Fälle geschrieben, sodass keine Verzweigungen mehr im Programm sind. Zweitens wurden unnötige verschachtelte Methodenaufrufe durch eine einzige Methode ersetzt, in der der gesamte Code steht. Für kleinere, häufig aufgerufene Methoden wurde im dritten Schritt versucht, denselben Effekt durch die Verwendung der inline-Anweisung zu erreichen. In der folgenden Tabelle ist die Veränderung der Rechenzeit des unlinearisierten Programms (T_{notLin}) zur linearisierten Version (T_{lin}) zu sehen:

Gitterpunkte	Matrizen	T_{notLin}	T_{lin}	$T_{lin}/T_{notLin} - 1$
10	L-Matrix	0.1599874	0.1591502	-0.5232912%
	H-Matrix	0.08616282	0.08476332	-1.624250%
	Beide	0.2461908	0.2439534	-0.908807315%
30	L-Matrix	1.41425	1.402736	-0.8141418%
	H-Matrix	0.7725324	0.7634898	-1.170514%
	Beide	2.186828	2.1662258	-0.9421043%
50	L-Matrix	3.946452	3.943718	-0.06927741%
	H-Matrix	2.195368	2.174534	-0.9489981%
	Beide	6.141868	6.118298	-0.3837595%

Tabelle 12: Laufzeitänderung durch Linearisierung

Die Linearisierung des Programms ergibt für alle betrachteten Diskretisierungen eine Performanzverbesserung, die allerdings für die Gesamtlaufzeit des Programms unter einem Prozent liegt.

6.6 Ersetzen von Matrix/Vektor-Operationen durch Eigen-Bibliothek

Sämtliche Operationen auf Matrizen und Vektoren, sei es das Invertieren oder Transponieren der Jakobimatrix, das Berechnen der Determinanten oder die notwendigen Matrix-Vektor Multiplikationen zum Berechnen der Einträge der L-Matrix, wurden zunächst mit selbst geschriebenen Methoden umgesetzt. Hierbei wurde bereits darauf geachtet, möglichst effiziente Methoden zu implementieren. Im Versuch, die Codelaufzeit weiter zu minimieren, wurde das Programm gänzlich umgeschrieben, um anstatt von Vektoren aus der std-library in Kombination mit selbst geschriebenen Methoden nun Matrizen aus der Eigen-Bibliothek, einer OpenSource Implementierung für effiziente Matrixoperationen, zu verwenden. Dadurch sollte getestet werden, ob die selbstgeschriebenen Funktionen im Vergleich zu einer alternativen, effizienzorientierten Implementierung zu Performanzeinbußen führen. In Tabelle 13 lässt sich der Unterschied zwischen der Laufzeit der Version mit selbstgeschriebenen Matrix- und Vektoroperationen T_{own} und der Laufzeit der Version, die die Eigen-Bibliothek verwendet T_{Eigen}, sehen:
Wie sich erkennen lässt, führt die abgeänderte Implementierung mit Nutzung von Matrixobjekten der Eigen-Bibliothek über alle Problemgrößen hinweg zu einem Performanznachlass. Dies kann mehrere Gründe haben. Einerseits sind die Matrizen, auf denen gerechnet wird, relativ klein mit Größen von 3x3 bis 8x8, wobei die einzigen auf Matrizen der Größe 8x8 ausgeührten Operationen die Multiplikation mit einem skalaren Wert und die anschliessende Addition mehrerer solcher Matrizen sind. Zudem sind sie vollständig besetzt. Somit bieten die Berechnungen wenig Ansatzpunkte

für eine Laufzeitoptimierung. Hinzu kommt der zusätzliche Overhead durch das Anlegen und Nutzen der Matrixobjekte, welcher bei gleicher Effizienz der genutzen Methoden für den Anstieg in der Laufzeit verantwortlich sein dürfte. Dieser ist aufgrund der höheren Komplexität der Matrixobjekte im Gegensatz zu den std-Vektoren höher als bei der vektorbasierten Implementierung. Somit ist die Variante mittels selbstgeschriebener Matrixoperationen hier performanter.

Gitterpunkte	Matrizen	T_{own}	T_{Eigen}	$T_{Eigen}/T_{own} - 1$
10	L-Matrix	0.1599874	0.1693042	+5.823459%
	H-Matrix	0.08616282	0.09200182	+6.776705%
	Beide	0.2461908	0.260943	+5.992181%
30	L-Matrix	1.41425	1.486688	+5.122008%
	H-Matrix	0.7725324	0.823977	+6.659215%
	Beide	2.186828	2.310708	+5.664826%
50	L-Matrix	3.946452	4.155752	+5.303498%
	H-Matrix	2.195368	2.34403	+6.771620%
	Beide	6.141868	6.499834	+5.828292%

Tabelle 13: Laufzeitänderung durch Nutzen der Eigen-Bibliothek

6.7 Simultanes Berechnen beider Matrizen

In der aktuellen Implementierung werden zunächst sämtliche lokale H-Matrizen und anschliessend sämtliche lokale L-Matrizen berechnet. Somit werden sämtliche Operationen, die bei der Berechnung beider Matrizen identisch sind, unnötigerweise für jedes einzelne Element doppelt ausgeführt. Dabei handelt es sich einerseits um das Auslesen der Eckpunktkoordinaten des jeweiligen Elements aus der UGBlocks-Diskretisierung und das Anlegen des entsprechenden Koordinatenvektors zur Übergabe an das Programm, welches die zugehörige lokale Matrix berechnet. Außerdem ist auch die Berechnung der Jakobimatrix mittels dieser Koordinaten identisch für die H-Matrix und die L-Matrix. Um diese theroetisch unnötigen doppelten Schritte einzusparen, müssten allerdings große Teile des UGBlocks-Codes grundsätzlich umgeschrieben werden, um die gleichzeitige Berechnung und Rückgabe zweier Matrizen zu ermöglichen. Dies liegt daran, dass die Klasse, in der die Berechnung der lokalen Steifigkeitsmatrizen aufgerufen wird, von Grund auf so konstruiert ist, dass sie nur eine globale Matrix zusammenzusetzen und zurückzugeben kann. Um dennoch einschätzen zu können, ob eine elementweise gemeinsame Berechnung zu einer Performanzverbesserung führen kann und wenn ja, welche Größenordnung diese hat, wurde die Dauer eines Aufrufs einer gekürzten Version von Calculate gemessen. In diesem Aufruf wird wie in der normalen Methode über sämtliche Elemente iteriert und die Koordinaten jedes Elements in einen Vektor geschrieben. Zusätzlich wird pro Element noch die Jakobimatrix berechnet. Wenn man die so gemessene Laufzeit nun mit der Gesamtlaufzeit der ursprünglichen, getrennten Berechnung beider Matrizen vergleicht, so erhält man eine Abschätzung der möglichen Laufzeitersparnis durch eine Implementierung, die auf einem Element gleichzeitig beide Matrizen berechnet. Die gemessenen Werte sind in Tabelle 14 und Tabelle 15 zu sehen, wobei T_{sep} die Zeit zur standardmäßigen, separaten Berechnung angibt und T_{saved} die einsparbare Rechendauer. Der Zusatz „own“ bedeutet hierbei, dass es sich um Messungen unter Nutzung der eigenen Implementierung handelt und der Zusatz „col“, dass das Experiment für die Implementierung mittels Colsamm durchgeführt wurde.

Gitterpunkte	$T_{sep,own}$	$T_{saved,own}$	$(T_{sep,own} - T_{saved,own})/T_{sep,own} - 1$
10	0.2439534	0.04447658	-18.23159%
30	2.1662258	0.384326	-17.74173%
50	6.118298	1.05147	-17.18566%

Tabelle 14: Potentielle Ersparnis durch gleichzeitiges Berechnen der Matrizen, eigene Implementierung

Gitterpunkte	$T_{sep,Col}$	$T_{saved,Col}$	$(T_{sep,Col} - T_{saved,Col})/T_{sep,Col} - 1$
10	0.187949	0.03871117	-20.59664%
30	1.617882	0.3339085	-20.63862%
50	4.603664	0.9297098	-20.19499%

Tabelle 15: Potentielle Ersparnis durch gleichzeitiges Berechnen der Matrizen, Implementierung mit Colsamm

Es ist eindeutig zu erkennen, dass unabhängig von der Wahl des Codes zur Berechnung der lokalen Steifigkeitsmatrizen eine deutliche Steigerung in der Performanz erreichbar ist, wenn man die Berechnung der beiden Matrizen miteinander verknüpft. Das doppelte Anlegen der Koordinatenvektoren der einzelnen Elemente bei getrennter Berechnung machen bei Nutzung von Colsamm laut dem durchgeführten Experiment circa 20 bis 21 Prozent der gesamten Laufzeit zur Berechnung beider Matrizen aus. Im Fall der selbst geschriebenen Implementierung ist potentiell eine Verkürzung der gesamten Rechendauer von ca. 17 bis 18 Prozent möglich. Die totale eingesparte Rechenzeit ist hier zwar höher, was durch das Einsparen der doppelten Berechnung der Jakobimatrizen erreicht wird, allerdings ist auch die gesamte Laufzeit höher, was prozentual zu einer geringeren potentiellen Ersparnis im Vergleich zu Colsamm führt. Insgesamt wurde durch das hier durchgeführte Experiment bestätigt, dass es möglich ist für Probleme, bei denen mehrere Matrizen zur Aufstellung eines Gleichungssystems berechnet werden müssen, sinnvoll sein kann, eine Anpassung des UGBlock-Codes vorzunehmen, die es erlaubt, sämtliche benötigten Matrizen gemeinsam zu berechnen anstatt nacheinander. Eine derartige Implementierung kann dann innerhalb des Programms, welches tatsächlich die lokalen Steifigkeitsmatrizen berechnet, weiter optimiert werden.

6.8 Performanzvergleich der besten Implementierung

Die performanteste im Rahmen dieser Bachelorarbeit angefertigte Implementierung zur Berechnung lokaler Steifigkeitsmatrizen war diejenige, die sowohl das doppelte Berechnen einzelner Ausdrücke einspart, als auch eine linearisierte Version des Codes verwendet. Sämtliche anderen verfolgten Ansätze, die in Kombination mit UGBlocks umgesetzt werden konnten, führten zu einer Verlängerung der Rechenzeit. In Tabelle 16 sind die Unterschiede in den Laufzeiten der ersten eigenen Implementierung T_{first}, der performantesten eigenen Implementierung T_{opt} und der Colsamm-Implementierung T_{col} zu sehen.

Wie sich sehen lässt, ist es gelungen, die Rechenzeit des Gesamtprogramms um ca. 13 bis 14 Prozent zu senken. Die Dauer der Berechnung der L-Matrix konnte sogar um ca. 17 bis 19 Prozent gesenkt werden. Allerdings ist auch die performanteste der selbst angefertigten Implementierungen noch um etwa 25 Prozent langsamer als die Implementierung durch Colsamm.

Gitterpunkte	Matrizen	T_{first}	T_{opt}	T_{col}	$T_{opt}/T_{first} - 1$	$T_{col}/T_{opt} - 1$
10	L-Matrix	0.1937482	0.1591502	0.10955	-17.85719%	-31.16565%
	H-Matrix	0.08712882	0.08476332	0.0783575	-2.714945%	-7.557301%
	Beide	0.2809168	0.2439534	0.187949	-13.15813%	-22.95701%
30	L-Matrix	1.718218	1.402736	0.9298768	-18.36099%	-33.70978%
	H-Matrix	0.7955212	0.7634898	0.687959	-4.026467%	-9.892837%
	Beide	2.513786	2.1662258	1.617882	-13.82616%	-25.31333%
50	L-Matrix	4.808702	3.943718	2.631772	-17.98789%	-33.26673%
	H-Matrix	2.247628	2.174534	1.971842	-3.252050%	-9.321169%
	Beide	7.05638	6.118298	4.603664	-13.29409%	-24.75581%

Tabelle 16: Vergleich der Laufzeit der langsamsten und der schnellsten Version sowie Colsamm

6.9 Anteil der Matrixberechnung an der Gesamtlaufzeit

Um die Relevanz der Performanz der Berechnung der lokalen Steifigkeitsmatrizen in Kontext zu setzen, wurde für alle drei genutzten Diskretisierungen ebenfalls die Dauer des Berechnens der Lösung der Poisson-Gleichung mit dem entstehenden Gleichungssystem gemessen. Als Löser wurde hier das konjugierte-Gradienten Verfahren verwendet, welches in dem genutzten Programmbeispiel von UGBlocks bereits implementiert war. Die Dauer der Lösung des Gleichungssystem kann anschließend mit der Dauer der Berechnung der zum Aufstellen des Gleichungssystems benötigten H-Matrizen und L-Matrizen $T_{lokStMatr}$ verglichen werden, um festzustellen, welchen prozentualen Anteil diese Berechnungen an der Gesamtdauer des Lösens der Poisson-Gleichung $T_{Poisson}$ auf dem betrachteten Gebiet haben.

Gitterpunkte	$T_{lokStMatr}$	$T_{Poisson}$	$T_{lokStMatr}/T_{Poisson}$
10	0.25064s	0.67804s	36.965%
30	2.1834s	22.861s	9.5507%
50	6.3234s	96.834s	6.5301%

Tabelle 17: Verhältnis der Berechnungsdauer der lokalen Steifigkeitsmatrizen zur Gesamtdauer des Lösens der Poissongleichung

Im Fall der Diskretisierung mit 10^3 Gitterpunkten macht die Berechnung der lokalen Steifigkeitsmatrizen noch über 36 Prozent der Gesamtdauer aus. Je mehr Gitterpunkte genutzt werden, desto geringer ist der prozentuale Anteil an der Gesamtlaufzeit. Somit rentiert sich der Zusatzaufwand des Anfertigens eines problemspezifischen, optimierten Programms zur Berechnung lokaler Steifigkeitsmatrizen in erster Linie für Diskretisierungen mit eher wenigen Gitterpunkten.

7 Zusammenfassung und Ausblick

In Rahmen dieser Bachelorarbeit wurde die Finite Elemente Methode am Beispiel der Poisson-Gleichung beschrieben und die Berechnung der lokalen Steifigkeitsmatrizen für Dreieckselemente, Viereckselemente, Tetraederelemente und Hexaederelemente implementiert. Hierfür wurden die mathematischen und numerischen Grundlagen einer effizienten Berechnung der lokalen Steifigkeitsmatrizen erklärt und die benötigten Formeln hergeleitet. Zudem wurden diverse Änderungen an der Implementierung vorgenommen, um die Laufzeit des Programms zu minimieren. Von diesen Variationen erbrachten nur zwei eine tatsächliche Laufzeitverbesserung, zusammen konnte eine Verkürzung der gesamten Berechnungsdauer von ca. 13 Prozent erreicht werden. Allerdings gelang es nicht, die Laufzeit der aktuell von UGBlocks genutzten Implementierung zur Berechnung lokaler Steifigkeitsmatrizen, Colsamm, zu unterbieten, da die beste erreichte Laufzeit des selbst geschriebenen Programms bei sämtlichen getesteten Diskretisierungen ca. 25 Prozent langsamer war als bei Nutzung von Colsamm. Zudem wurde festgestellt, dass die Laufzeit des Codes zur Berechnung der lokalen Steifigkeitsmatrizen im Bezug auf die gesamte Rechendauer zur Lösung der Poisson-Gleichung auf einem Gebiert bei Diskretisierungen mit wenigen Elementen deutlich stärker ins Gewicht fällt als bei Diskretisierungen mit vielen Elementen. Auch wurde experimentell festgestellt, dass eine Änderung des UGBlocks-Codes, die es erlauben würde sämtliche zur Lösung einer DGL nötigen Matrizen gleichzeitig zu berechnen, unabhängig von der genutzten Implementierung zur Berechnung der lokalen Steifigkeitsmatrizen eine deutliche Laufzeitverbesserung mit sich bringen würde. Ein sinnvoller Ansatzpunkt für eine weitere Arbeit wäre somit, eine derartige Modifizierung für UGBlocks vorzunehmen. Ebenfalls interessant wäre eine Untersuchung der auf Expression Templates basierten Implementierung von Colsamm, um festzustellen, warum diese so effizient ist. Während die Aufgabenstellung dieser Bachelorarbeit das Anfertigen einer von Grund auf selbst geschriebenen Implementierung war, könnte man stattdessen versuchen, das Programm Colsamm zu erweitern und beispielsweise für das Lösen der Poisson-Gleichung optimieren.

Literatur

[Ale07] Jochen Härdtlein und Alexander Linke und Christoph Pflaum. *Fast Expression Templates - Enhancements for High Performance C++ Libraries, https://www10.cs.fau.de/publications/reports/TechRep*$_2$0 07.*pdf*. 2007.

[Boa21] OpenMP Architecture Review Board. *https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf*. 2021.

[Här07] Jochen Härdtlein. *https://www.cs10.tf.fau.de/research/software/expde/colsamm/*. 2007.

[Pfl] Christoph Pflaum. *Simulation und wissenschaftliches Rechnen II, https://www. cs10.tf.fau.de/files/2018/06/pflau script-siwir2.pdf, zuletzt abgerufen am: 28.04.2023*.

[Pfl10] Christoph Pflaum. *https://www.cs10.tf.fau.de/research/software/expde/ unstructured-block-grids/*. 2010.

[Rei19] Frank Rieg und Reinhard Hackenschmidt und Bettina Alber-Laukant. *Finite Elemente Analyse für Ingenieure (6. Aufl.)* Hanser, 2019.