

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

Implementierung einer Finite-Elemente-Methode auf der Grafikkarte

BACHELORARBEIT

*zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.) im Studiengang Mathematik*

vorgelegt von Markus Pawellek

geboren am 7. Mai 1995 in Meiningen
Matrikelnummer: 144645

Erstgutachter: Gerhard Zumbusch

Jena, 31. August 2018

Zusammenfassung

Der Gegenstand dieser Arbeit ist es, die Finite-Elemente-Methode (FEM) effizient auf dem Grafikprozessor (GPU) eines Computers am Beispiel der zweidimensionalen Wellengleichung zu implementieren. Zugrundeliegende Berechnungsgebiete werden dabei durch eine ausreichend große Anzahl von Dreiecken, deren Eckpunkten und deren Kanten diskretisiert. Das daraus folgende zu lösende System von algebraischen Gleichungen stellt für praxisnahe Probleme jedoch eine Herausforderung dar. Durch die Art der erforderlichen Berechnungen und durch die notwendige Rechenleistung ermöglicht die Verwendung der GPU allerdings einen extremen Anstieg der Performance. Für die Programmierung der FEM auf der GPU werden das CSR-Format für dünnbesetzte Matrizen und die iterative CG-Methode zur Lösung von linearen Gleichungssystemen implementiert. Unter Zuhilfenahme von CUDA werden die angegebenen Datenstrukturen und Algorithmen sowohl auf einer NVIDIA Grafikkarte als auch auf der CPU konstruiert. Die erhaltenen Lösungen werden zudem visualisiert und getestet.

Inhaltsverzeichnis

1 Einleitung	1
2 Grundlagen	3
2.1 Modelprobleme	3
2.1.1 Berechnungsgebiet	3
2.1.2 Poisson-Gleichung	4
2.1.3 Wellengleichung	6
2.2 Aufbau und Funktionsweise des Grafikprozessors	8
2.2.1 GPU Computing mit CUDA	8
2.2.2 Architektur einer modernen GPU	10
3 Methoden	11
3.1 Finite-Elemente-Methode	11
3.1.1 Konstruktion der schwachen Formulierung	11
3.1.2 Diskretisierung des Berechnungsgebietes	14
3.1.3 Wahl der Basisfunktionen	15
3.1.4 Konstruktion der algebraischen Formulierung	18
3.1.5 Lösung des algebraischen Systems	22
3.2 Numerische Methoden dünnbesetzter Matrizen	22
3.2.1 Datenstrukturen und das CSR-Format	22
3.2.2 Lösung dünnbesetzter Systeme und das CG-Verfahren	24
4 Implementierung	27
4.1 Repräsentation des Berechnungsgebietes	27
4.2 Aufbau des Systems	33
4.3 Konstruktion der Systemmatrizen	35
4.4 Berechnung eines Zeitschrittes	39
4.5 Visualisierung	43
5 Simulation und Ergebnisse	47
6 Fazit und Aussicht	51
Literatur	52
A Wärmeleitungsgleichung	i
B ELLPACK-Format	v

Abbildungsverzeichnis

1	Einfache Simulation der Wellengleichung	1
2	Beispiel eines zweidimensionalen Berechnungsgebietes	4
3	Grundlegende Architektur einer CUDA-fähigen GPU	9
4	Schematisches Vorgehen der Finite-Elemente-Methode	11
5	Diskretisierung eines eindimensionalen Berechnungsgebietes	14
6	Diskretisierung eines zweidimensionalen Berechnungsgebietes	14
7	Eindimensionale Hutfunktionen	15
8	Zweidimensionale Hutfunktion	16
9	Beispiele verschiedener Berechnungsgebiete	31
10	Subdivision-Schema	31
11	Subdivision von Testgebieten	32
12	Wellensimulation auf einem Kreis	44
13	Wellensimulation auf dem Testgebiet	44
14	Wellensimulation auf einem quadratischen Ring	45
15	Wellensimulation auf einem quadratischen Gebiet	45
16	Diagramm des GPU-Speedups	48
17	Wellensimulation auf einem Torus	51
18	Wellensimulation auf einer Kugel	52

Symboltabelle

Symbol	Definition
$x \in A$	x ist ein Element der Menge A .
$A \subset B$	A ist eine Teilmenge von B .
$A \cap B$	$\{x \mid x \in A \text{ und } x \in B\}$ für Mengen A, B — Mengenschnitt
$A \cup B$	$\{x \mid x \in A \text{ oder } x \in B\}$ für Mengen A, B — Mengenvereinigung
$A \setminus B$	$\{x \in A \mid x \notin B\}$ für Mengen A, B — Differenzmenge
$A \times B$	$\{(x, y) \mid x \in A, y \in B\}$ für Mengen A und B — kartesisches Produkt
\emptyset	$\{\}$ — leere Menge
\mathbb{N}	Menge der natürlichen Zahlen
\mathbb{N}_0	$\mathbb{N} \cup \{0\}$
\mathbb{R}	Menge der reellen Zahlen
\mathbb{R}^n	Menge der n -dimensionalen Vektoren
$\mathbb{R}^{n \times n}$	Menge der $n \times n$ -Matrizen
$f: X \rightarrow Y$	f ist eine Funktion mit Definitionsbereich X und Wertebereich Y
$\partial\Omega$	Rand einer Teilmenge $\Omega \subset \mathbb{R}^n$
σ	Oberflächenmaß
λ	Lebesgue-Maß
$\int_{\Omega} f d\lambda$	Lebesgue-Integral von f über der Menge Ω
$\int_{\partial\Omega} f d\sigma$	Oberflächen-Integral von f über der Menge $\partial\Omega$
∂_i	Partielle Ableitung nach der i . Koordinate
∂_t	Partielle Ableitung nach der Zeitkoordinate
∂_i^2	Zweite partielle Ableitung nach i
∇	$\begin{pmatrix} \partial_1 & \partial_2 \end{pmatrix}^T$ — Nabla-Operator
Δ	$\partial_1^2 + \partial_2^2$ — Laplace-Operator
$C^k(\Omega)$	Menge der k -mal stetig differenzierbaren Funktion auf Ω
$L^2(\Omega)$	Menge der quadrat-integrierbaren Funktionen auf Ω
$H^1(\Omega)$	Sobolevraum
$f _{\partial\Omega}$	Einschränkung der Funktion f auf $\partial\Omega$
$\langle x, y \rangle$	Euklidisches Skalarprodukt
$[a, b]$	$\{x \in \mathbb{R} \mid a \leq x \leq b\}$
(a, b)	$\{x \in \mathbb{R} \mid a < x < b\}$
$[a, b)$	$\{x \in \mathbb{R} \mid a \leq x < b\}$
$u(\cdot, t)$	Funktion \tilde{u} mit $\tilde{u}(x) = u(x, t)$
A^T	Transponierte der Matrix A
id	Identitätsabbildung
$a := b$	a wird durch b definiert
$f \circ g$	Komposition der Funktionen f und g
$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$	Determinante der angegebenen Matrix
$\text{span}\{\dots\}$	Lineare Hülle der angegebenen Menge
$ A $	Anzahl der Elemente in der Menge A

1 Einleitung

In der Physik werden wichtige Vorgänge der Natur oder technischer Prozesse häufig durch partielle Differentialgleichungen beschrieben. Gerade in der industriellen Forschung und den Ingenieurwissenschaften benötigt man Lösungen dieser Gleichungen, um neue Bauteile und Verfahren, die gewissen Bedingungen genügen müssen, zu konstruieren. Als Beispiel sei hier die Wärmeleitungsgleichung genannt, deren Lösung es ermöglicht, die Temperaturverteilung und den Wärmevertrag eines solchen Bauteils zu bestimmen. Die Lösungen weisen auf die Schwächen und Stärken des Bauteils hin, ohne dieses real konstruieren zu müssen. [8, 16]

Die partiellen Differentialgleichungen zusammen mit der Geometrie der Bauteile und den Materialeigenschaften praxisnaher Probleme sind jedoch meistens zu kompliziert, um sie analytisch zu lösen. Entweder es existiert nicht einmal eine geschlossene Lösung oder sie wäre viel zu kompliziert. Aus diesem Grund beschränkt man sich in diesen Bereichen auf verschiedene numerische Verfahren, die die Gleichungen, die Geometrie und Materialeigenschaften diskretisieren und in ein System algebraischer Gleichungen transformieren. Die erhaltenen Ergebnisse des diskretisierten Systems stellen zwar nur eine Approximation der eigentlichen Lösung dar, können diese jedoch meistens beliebig genau annähern. Typische Verfahren zum Lösen von partiellen Differentialgleichungen stellen die Finite-Differenzen-Methode, die Finite-Volumen-Methode und die Finite-Elemente-Methode dar. In Abbildung 1 wird die numerische Simulation der Wellengleichungen über mehrere Zeitschritte hinweg anhand eines Beispiels demonstriert, indem die Finite-Elemente-Methode verwendet wurde. [4, 8, 13, 16]

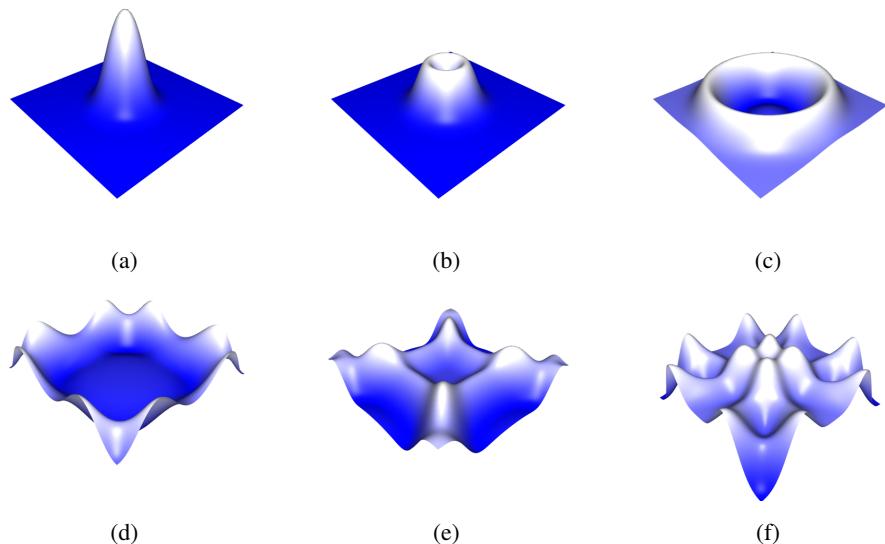


Abbildung 1: Die Abbildung zeigt die Evolution einer numerischen Lösung der Wellengleichung auf einem quadratischen zweidimensionalen Gebiet. Für die Berechnung wurde die hier implementierte Finite-Elemente-Methode verwendet.

Die Finite-Elemente-Methode stellt eines der allgemeinsten Verfahren zum numerischen Lösen von partiellen Differentialgleichungen dar. Einer ihrer größten Vorteile besteht darin, komplexe Geometrien und Randbedingungen durch die Art der Diskretisierung beliebig genau beschreiben zu können. Gerade deshalb tendiert die Finite-Elemente-Methode jedoch dazu, enorme Rechenleistungen zu benötigen, um die resultierenden algebraischen Gleichungen zu lösen. Allerdings konnte sie aufgrund ihrer im Vergleich zu anderen Verfahren späten Entwicklung durch ein mathematisches Vorgehen computergerecht formuliert werden, sodass Implementierungen dieser Methode die Hardware des Computers im Allgemeinen effizient ausnutzen. Zudem ist die Anwendung der Finite-Elemente-Methode nicht auf ein konkretes physikalisches Gebiet beschränkt. Obwohl sie zunächst nur für die Strukturberechnung von Flugzeugflügeln in der Luft- und Raumfahrtindustrie verwendet wurde, findet sie heute in vielen Bereichen der Physik, wie zum Beispiel bei der Berechnung des Wärmetransports, der Strömungssimulation, der Festigkeits- und Verformungsuntersuchung von Festkörpern, der gekoppelten Feldberechnung und bei Wettervorhersagen, ihre Anwendung. [4, 8, 13]

Das allgemeine Ziel besteht nun darin immer komplexer werdende Probleme zu lösen, indem man in das zugrundeliegende Modell neue physikalische Effekte mit einbezieht oder die Diskretisierung des Modells verfeinert, um die genannten Effekte besser aufzulösen. Folglich ist es erforderlich, die Performance eines Programms immer weiter zu steigern. Dies erreicht man heutzutage durch dessen Parallelisierung. Aufgrund der computergerechten Formulierung der Finite-Elemente-Methode tritt bei ihr eine gewisse Datenparallelität auf. Diese Eigenschaft kann sehr gut durch den Grafikprozessors (GPU) eines Computers ausgenutzt werden. Die GPU ist ein massiv-paralleler Prozessor, dessen Prozessorkerne auf die Ausnutzung von Datenparallelität innerhalb einer Gruppe von Threads spezialisiert sind. Ein Algorithmus, der diese Parallelität nicht aufweist, lässt sich im Allgemeinen nur sehr ineffizient auf der GPU implementieren und sollte durch den Hauptprozessor (CPU) eines Computers, dessen Prozessorkerne für wesentlich komplexere Aufgaben entwickelt wurden, ausgeführt werden. Umgekehrt ist die GPU in der Lage, Algorithmen mit dieser Eigenschaft um ein Vielfaches zu beschleunigen. Gerade bei der Finite-Elemente-Methode sollte also die Implementierung auf der GPU diverse Vorteile mit sich bringen. [7, 11, 15]

2 Grundlagen

In den folgenden Abschnitten wird eine grundlegende Einführung und Definition der Strukturen und Verfahren gegeben, die für den weiteren Verlauf dieser Arbeit von Bedeutung sind. Viele dieser Themen können hier nur angerissen werden, da ihre komplette Behandlung den Rahmen und das Ziel des Themas sprengen würden. Für eine genauere Einführung in die einzelnen Themengebiete, wird dem Leser geraten, sich mit den genannten Quellen auseinanderzusetzen.

2.1 Modelprobleme

Für die Implementierung einer Finite-Elemente-Methode benötigt man zunächst ein mathematisches Modell, welches durch ein System partieller Differentialgleichungen beschrieben wird. Um sich jedoch nicht auf die Anwendung physikalischer Gesetze und auf das Verstehen in den darin resultierenden Modellen fokussieren zu müssen, werden in dieser Arbeit nur die einfachsten physikalisch basierten partiellen Differentialgleichungen als Grundlage vorausgesetzt. Die Poisson-, Wellen- und Wärmeleitungsgleichung beschreiben grundlegende Phänomene in der Elektrodynamik, Mechanik und Thermodynamik. Bei diesen drei Gleichungen handelt es sich um eine elliptische, eine hyperbolische und eine parabolische lineare partielle Differentialgleichung zweiter Ordnung in simpelster Form. Gerade deswegen kann man durch diese Gleichungen Modelprobleme formulieren, an denen das Vorgehen der Finite-Elemente-Methode unkompliziert erläutert werden kann, die aber auch komplex genug sind, um das Verfahren auch auf schwierigere Modelle anwenden zu können. [1, 8, 16]

Um die Visualisierung und die Konstruktion von Testgebieten zu erleichtern, werden die Gleichungen nur mit zwei Raumdimensionen behaftet. Dies ändert nichts Grundlegendes an der Implementierung und dient zudem der besseren Anschaulichkeit.

2.1.1 Berechnungsgebiet

Für die numerische Behandlung von partiellen Differentialgleichungen in zwei räumlichen Dimensionen ist eine genauere mathematische Betrachtung von Nutzen. Hierfür soll zunächst das räumliche Definitionsgebiet exakt definiert werden. Es wird sich hier auf [16, S. 39 f] und [1] bezogen. Die folgende Definition ist nicht im Sinne einer verallgemeinerten Theorie zu verstehen. Sie wurde in Hinsicht auf deren Verwendung für die hier gestellten Modelprobleme konstruiert und angepasst. Für komplexere Probleme lässt sich die Definition nach Belieben abändern, um sie für die Anwendung auf das entsprechende Problem zu optimieren.

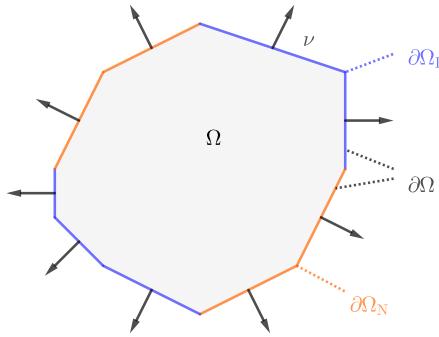


Abbildung 2: Die Abbildung zeigt ein schematisches Beispiel für ein zweidimensionales Berechnungsgebiet Ω . Der Rand $\partial\Omega$ teilt sich dabei in $\partial\Omega_D$, auf dem Dirichlet-Randbedingungen herrschen, und $\partial\Omega_N$, auf dem Neumann-Randbedingungen gelten sollen, ein. ν beschreibt eine Funktion, die für fast alle Punkte des Randes $\partial\Omega$ deren äußere Normale angibt.

DEFINITION 2.1: (Berechnungsgebiet)

Es sei $\Omega \subset \mathbb{R}^2$ ein beschränktes Lipschitz-Gebiet mit einem polygonalen Rand $\partial\Omega$. Die Funktion $\nu: \partial\Omega \rightarrow \mathbb{R}^2$ beschreibe fast überall in $\partial\Omega$ die äußere Normale von Ω . Weiterhin sei $\partial\Omega_D$ eine abgeschlossene Teilmenge des Randes $\partial\Omega$ mit positiver Länge und $\partial\Omega_N$ deren Komplement.

$$\partial\Omega_D \subset \partial\Omega , \quad \sigma(\partial\Omega_D) > 0 , \quad \partial\Omega_N := \partial\Omega \setminus \partial\Omega_D$$

In diesem Falle ist das Tupel $(\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ ein Berechnungsgebiet. Man nennt $\partial\Omega_D$ den Dirichlet-Rand, $\partial\Omega_N$ den Neumann-Rand und ν die äußere Normale von Ω .

Abbildung 2 veranschaulicht die Definition eines Berechnungsgebietes anhand eines schematischen Beispiels. Im Bezug auf die Definition sollen auf der Menge $\partial\Omega_D$ später die Dirichlet-Randbedingungen und auf deren Komplement $\partial\Omega_N$ Neumann-Randbedingungen gefordert werden. Die Forderung, dass es sich bei $\partial\Omega$ um einen polygonalen Rand handeln muss, vereinfacht die noch kommende Diskretisierung des Berechnungsgebietes und der zugehörigen Funktionenräume.

2.1.2 Poisson-Gleichung

Die Poisson-Gleichung stellt eine wichtige Grundlage für die Wellen- und Wärmeleitungsgleichung dar. Sie ist eine zeitunabhängige, elliptische lineare partielle Differentialgleichung zweiter Ordnung. Ihre Kernkomponente stellt die Verwendung des Laplace-Operators Δ dar, der in der Physik in fast allen Teilgebieten eine elementare Rolle einnimmt. Gerade aus diesem Grund ist die Poisson-Gleichung als ein Test für

die Implementierung einer Finite-Elemente-Methode ideal geeignet.

Zu Beginn soll die klassische Variante betrachtet werden, wie es auch in [16, S. 46] geschehen ist. Es seien $\Omega \subset \mathbb{R}^2$ eine nichtleere, offene, zusammenhängende Teilmenge und $f \in C^1(\Omega)$ eine stetige Funktion. Eine klassische Lösung $u \in C^2(\Omega)$ der Poisson-Gleichung erfüllt diese dann punktweise für alle $x \in \Omega$.

$$-\Delta u(x) := -(\partial_1^2 + \partial_2^2) u(x) = f(x)$$

Bereits in der Theorie erweist sich allerdings dieser klassische Lösungsbegriff als nicht ausreichend. Handelt es sich bei f um eine nicht-stetige Funktion, so kann u die Gleichung entweder nicht punktweise erfüllen oder nicht der Familie $C^2(\Omega)$ zweimal stetig differenzierbarer Funktionen angehören. Der Ausweg besteht darin einen verallgemeinerten schwachen Lösungsbegriff auf der Basis von Distributionen und Sobolevräumen einzuführen [16]. Diese erweisen sich dann auch in der numerischen Mathematik als ausreichend und dienen als Grundlage für die Formulierung der Finite-Elemente-Methode [8]. Über die Theorie der Distributionen und Sobolevräume können jedoch komplett Bücher geschrieben werden. Da man dies auch getan hat, soll hier für eine detailliertere Behandlung des Themas auf [16] verwiesen werden.

Die folgende Definition gibt die erste verallgemeinerte Form des Poisson-Problems an. Es wurde sich auch hier auf [1] und [16] gestützt, wobei diverse Notationen angepasst wurden.

DEFINITION 2.2: (Poisson-Problem)

Es seien $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ ein Berechnungsgebiet und die folgenden Funktionen gegeben.

$$f \in L^2(\Omega), \quad u_D \in H^1(\Omega), \quad u_N \in L^2(\partial\Omega_N)$$

Eine Funktion $u \in H^1(\Omega)$ nennt man eine schwache Lösung der Poisson-Gleichung, wenn die folgenden Gleichungen gelten.

$$-\Delta u = f \quad (\text{Poisson-Gleichung})$$

$$u|_{\partial\Omega_D} = u_D|_{\partial\Omega_D} \quad (\text{Dirichlet-Randbedingungen})$$

$$\langle \nabla u|_{\partial\Omega_N}, \nu \rangle = u_N \quad (\text{Neumann-Randbedingungen})$$

Das Tupel $([\Omega], f, u_D, u_N, u)$ wird dann auch Poisson-Problem genannt.

Die Funktion f ist auch bekannt als die Volumenkraft. Die Funktionen u_D und u_N stellen, wie bereits in der Definition erwähnt, die Dirichlet- und Neumann-Randbedingungen dar. Die Ableitungen in den Gleichungen der Definition sind im

Sinne der Distributionen zu verstehen. Diese Formulierung ist nun aber nicht besonders geeignet für die Konstruktion der Finite-Elemente-Methode. Für diese soll hier die schwache Formulierung verwendet werden, die sich aus der Anwendung des Gaußschen Satzes und der Berechnungsformel für die Ableitung von Distributionen ergibt [16, S. 62 ff].

DEFINITION 2.3: (Schwache Formulierung des Poisson-Problems)

Es seien $([\Omega], f, u_D, u_N, u)$ ein Poisson-Problem und die folgenden Definitionen gegeben.

$$v \in H_D^1(\Omega) := \{w \in H^1(\Omega) \mid w|_{\partial\Omega_D} = 0\}$$

$$v := u - u_D$$

Die Eigenschaft, dass es sich bei u um eine schwache Lösung handelt, ist dazu äquivalent, dass für alle $w \in H_D^1(\Omega)$ das Folgende gilt.

$$\int_{\Omega} \langle \nabla v, \nabla w \rangle \, d\lambda = \int_{\Omega} fw \, d\lambda + \int_{\partial\Omega_N} u_N w \, d\sigma - \int_{\Omega} \langle \nabla u_D, \nabla w \rangle \, d\lambda$$

Man nennt diese Gleichung die schwache Formulierung des Poisson-Problems.

Bei der hier notierten schwachen Formulierung wurde das Vorgehen aus [1] gewählt. Dieses erlaubt eine direkte Einarbeitung der Dirichlet-Randbedingungen für eine leichtere numerische Behandlung der noch folgenden algebraischen Gleichungen. Setzt man die Dirichlet- und Neumann-Randbedingungen identisch Null, so entsteht der folgende Spezialfall für alle $w \in H_D^1(\Omega)$, der den Formeln aus [16, S. 63] entspricht.

$$\int_{\Omega} \langle \nabla u, \nabla w \rangle \, d\lambda = \int_{\Omega} fw \, d\lambda$$

2.1.3 Wellengleichung

Wie schon die Poisson-Gleichung ist auch die Wellengleichung die einfachste ihrer Art. Sie ist eine zeitabhängige, hyperbolische, partielle Differentialgleichung zweiter Ordnung, die die Ausbreitung von Wellen auf dem zugrundeliegenden Berechnungsgebiet beschreibt. Die Wellengleichung, ob nun homogen oder inhomogen, spielt in praktisch jedem Teilgebiet der Physik eine große Rolle. Umso wichtiger ist die Fähigkeit, Wellen, die diese Gleichung lösen, auf komplexen Gebieten simulieren und berechnen zu können. Die Behandlung des zweidimensionalen Falls lässt sich mühelos auf den dreidimensionalen Fall ausweiten [16, S. 229 ff].

Für die Wellengleichung wählt man den Ansatz eines zeitlich konstanten Berechnungsgebietes. Hierbei sei bemerkt, dass die Länge des Dirichlet-Randes nicht notwendigerweise größer Null sein muss. Die Wellengleichung ist auch lösbar unter der Verwendung eines reinen Neumann-Randes. Um für die Wellengleichung eine eindeutige Lösung zu erlangen, müssen zwei Anfangsfunktionen vorgegeben werden. Diese Tatsache resultiert aus dem Vorkommen der zweiten zeitlichen Ableitung in der Wellengleichung. Die folgende Definition beschreibt dieses Vorgehen im Detail. [16]

DEFINITION 2.4: (Wellengleichung)

Es seien $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ ein Berechnungsgebiet und die folgenden Funktionen gegeben.

$$\begin{aligned} f &\in L^2(\Omega \times [0, \infty)) \\ u_0 &\in H^1(\Omega) & u_D &\in H^1(\Omega \times [0, \infty)) \\ u_1 &\in H^1(\Omega) & u_N &\in L^2(\partial\Omega_N \times [0, \infty)) \end{aligned}$$

Eine Funktion $u \in H^1(\Omega \times [0, \infty))$ nennt man eine schwache Lösung der Wellengleichung, wenn die folgenden Gleichungen für alle $t \in [0, \infty)$ gelten.

$$\begin{aligned} \partial_t^2 u - \Delta u &= f && \text{(Wellengleichung)} \\ u(\cdot, 0) &= u_0 && \text{(Anfangswerte)} \\ \partial_t u(\cdot, 0) &= u_1 && \text{(Anfangswerte)} \\ u(\cdot, t)|_{\partial\Omega_D} &= u_D(\cdot, t)|_{\partial\Omega_D} && \text{(Dirichlet-Randbedingungen)} \\ \langle \nabla u(\cdot, t)|_{\partial\Omega_N}, \nu \rangle &= u_N(\cdot, t) && \text{(Neumann-Randbedingungen)} \end{aligned}$$

Das Tupel $([\Omega], f, u_0, u_1, u_D, u_N, u)$ wird dann auch Wellenproblem genannt.

Zu bemerken sei hier, dass für die Konstruktion der Finite-Elemente-Methode die rigorose schwache Formulierung der Wellengleichung keine direkte Rolle spielt. Beim Übergang zur Diskretisierung erweist es sich als praktisches Vorgehen, jegliche Zeitableitung zuvor durch einen diskreten Differenzenquotienten zu ersetzen, sodass es sich bei dem resultierenden Gleichungssystem streng genommen nicht mehr um eine zeitabhängige Differentialgleichung handelt [1, 16]. Demzufolge soll hier auf die Angabe der schwachen Formulierung verzichtet werden, da diese für die Konstruktion der Finite-Elemente-Methode nicht im Vordergrund steht. Nach [16, S. 236 ff] existiert für die hier angegebene Wellengleichung eine eindeutige Lösung, die durch das Galerkin-Verfahren beschrieben wird.

Ein typischer Spezialfall der Wellengleichung setzt alle Randterme und äußereren Einwirkungen identisch zu Null. Stellt man sich die Welle im Zweidimensionalen wie ein Tuch vor, so bedeutet ein Dirichlet-Rand mit dem Wert Null, dass das Tuch an diesem Rand festgehalten wird, und ein Neumann-Rand mit dem Wert Null, dass das Tuch an diesem Punkt frei schwingen kann.

2.2 Aufbau und Funktionsweise des Grafikprozessors

Der Grafikprozessor (GPU, engl.: *graphics processing unit*) eines Computers stellt eine weitere Prozessorart gegenüber dem Hauptprozessor (CPU, engl.: *central processing unit*) dar. Im Gegensatz zur CPU ist der Aufbau und die Funktionsweise der GPU auf konkrete Aufgaben spezialisiert. In ihrer einfachsten Form generiert die GPU zweidimensionale und dreidimensionale Grafiken, Bilder und Videos, die grafische Benutzeroberflächen, Computerspiele, Video- und Bildbearbeitung ermöglichen. Die moderne GPU ist ein massiv paralleler Multiprozessor optimiert für visuelle Berechnungen (engl.: *visual computing*). Um dem Benutzer eine visuelle Interaktion in Echtzeit zu bieten, besitzt die GPU eine vereinheitlichte Grafik- und Prozessorarchitektur (engl.: *graphics and computing architecture*). Diese dient sowohl als programmierbarer Grafikprozessor sowie als skalierbare parallele Plattform. Auf herkömmlichen Systemen werden zudem CPUs mit GPUs verbunden, um ein sogenanntes heterogenes System zu bilden, welches die Vorteile der jeweiligen Prozessorarten ausnutzt. [11, S. A3]

Gerade bei parallelisierbaren Problemen mit kohärenten und linearen Speicherzugriffen, die auf Matrix- und Vektoroperationen aufbauen, stellt die Verwendung der GPU einen enormen Anstieg der Effizienz dar. Aus Abschnitt 3.1 wird ersichtlich, dass gerade ein Programm, welches auf der Finite-Elemente-Methode basiert, durch die GPU stark beschleunigt werden könnte. Auch die Auflösung der Diskretisierung des zugrundeliegenden Berechnungsgebietes könnte erhöht werden, da die GPU mit der Anzahl der finiten Elemente skalieren kann.

2.2.1 GPU Computing mit CUDA

»GPU Computing« ist das Ausnutzen der GPU für Berechnungen mithilfe einer parallelen Programmiersprache und einer Programmierschnittstelle (API, engl.: *application programming interface*). Dies steht im Gegensatz zum ursprünglichen GPGPU-Modell (engl.: *general purpose computation on GPU approach*), in welchem die GPU über traditionelle Graphik-APIs, wie zum Beispiel durch OpenGL GLSL, dazu gebracht wurde, Berechnungen durchzuführen, die nicht die Ausgabe eines Bildes auf dem Bildschirm zum Ziel hatten. Das von NVIDIA entwickelte CUDA ist ein skalierbares paralleles Programmiermodell, eine parallele Programmiersprache und eine API, die es dem Programmierer erlauben, die typische Grafikschnittstelle der GPU zu umgehen und diese direkt durch die Sprachen C und C++ zu programmieren [11,

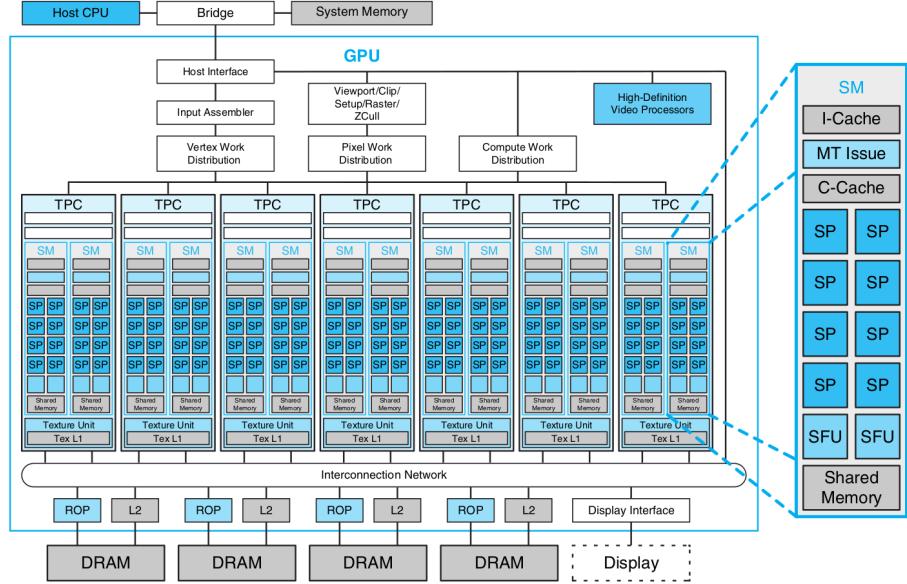


Abbildung 3: Die Abbildung zeigt die grundlegende vereinheitlichte Architektur einer CUDA-fähigen GPU anhand eines Schemas. Es handelt sich dabei nur um Beispiele. [7, S. 9]

S. A5]. Das CUDA-Programmiermodell arbeitet nach dem Prinzip »Single-Program Multiple Data« (SPMD). Der Programmierer schreibt hierbei ein Programm für einen einzelnen Thread, welches dann durch die GPU durch mehrere Threads instanziert und ausgeführt wird [11, S. A5].

Für einen CUDA-Programmierer besteht ein Computersystem aus dem sogenannten »Host«, welcher in den meisten Fällen die CPU darstellt, und mindestens einer »Device«, die die GPU repräsentiert [7, S. 39]. Ein CUDA-Programm teilt sich nun in mehrere Phasen, in denen Code entweder auf dem Host oder der Device ausgeführt wird, ein. Code, der eine geringe Datenparallelität aufweist, wird dabei zumeist auf dem Host ausgeführt [7]. Umgekehrt sollte Code mit einer hohen Datenparallelität auf der GPU beziehungsweise der Device ausgeführt werden, um deren Vorteile zu nutzen [7]. Um bei der Ausführung des Quelltextes zwischen Host und Device wechseln zu können, bietet CUDA diverse Schnittstellen an. Ein CUDA-Programm startet zunächst auf dem Host, der durch den Aufruf eines entsprechenden »Kernels« in die Lage versetzt wird, Code auf mehreren Prozessoren der GPU auszuführen. Daten die zunächst nur im Arbeitsspeicher (RAM, engl.: *random-access memory*) lagen und damit nur durch die CPU verwendbar waren, können durch CUDA bereit gestellte Befehle über den »PCI-Express Bus« (PCIe), der eine Datenverbindung zwischen der CPU und GPU bildet, in den dynamischen Arbeitsspeicher (DRAM, engl.: *dynamic random-access memory*) der GPU übertragen werden. Zudem ist es dem Host auch möglich, Daten vom DRAM der Device anzufordern. Auch diese werden dann über den PCIe in den RAM übertragen. [15]

2.2.2 Architektur einer modernen GPU

Vereinheitlichte GPU-Architekturen basieren auf einer parallelen Anordnung von vielen programmierbaren Prozessoren, die auch die typischen Berechnungen für visuelle Ausgaben vollführen. Im Vergleich zu einer mehrkernigen CPU ist die Architektur einer GPU darauf ausgerichtet, extrem viele parallele Threads effizient auf einer großen Anzahl von Prozessorkernen auszuführen. In einer GPU wird diese Effizienz erreicht, indem man einfachere Prozessorkerne als in der CPU verwendet. Diese sind auf die Ausnutzung der Datenparallelität innerhalb einer Gruppe von Threads optimiert, sodass im Gegensatz zu einer CPU kleinere Caches und mehr Transistoren für arithmetische Einheiten verwendet werden können. [11, S. A11]

Abbildung 3 skizziert die Architektur einer typischen CUDA-fähigen NVIDIA GPU. Sie besteht aus einem Feld von 112 »Streaming Processor Cores« (SP), die wiederum in 14 »Multithreaded Streaming Multiprocessors« (SM) eingeteilt sind. Jeder SP ist dazu in der Lage, bis zu 100 parallele Threads in seiner Hardware zu organisieren und zu speichern. Die Prozessoren sind über ein Verbindungsnetzwerk mit Partitionen des dynamischen Arbeitsspeichers (DRAM, engl.: *dynamic random-access memory*) verbunden. Zudem besitzt jeder SM »Special Function Units« (SFU), »Instruction Caches«, »Constant Caches«, »Multithreaded Instruction Units« und »Shared Memory«. Durch Hinzunahme von SMs und DRAM-Partitionen erhält man ein skalierbares Verfahren, um schwächere und stärkere GPU-Konfigurationen zu konstruieren.

3 Methoden

3.1 Finite-Elemente-Methode

Die Finite-Elemente-Methode ist, wie bereits in Abschnitt 1 erwähnt, ein allgemeines numerisches Verfahren, um partielle Differentialgleichungen approximativ zu lösen. Genauer gesagt, handelt es sich um eine spezielle Form des Ritz-Galerkin-Verfahrens. Die Finite-Elemente-Formulierung eines Problems resultiert in einem System von algebraischen Gleichungen. Um ein Problem zu lösen, unterteilt es das Berechnungsgebiet in kleinere und einfach zu behandelnde Teile, die auch finite Elemente genannt werden. Die einfachen Systeme von algebraischen Gleichungen der einzelnen finiten Elementen werden dann in ein großes System von Gleichungen, welches das gesamte Problem modelliert, assembled. Durch die Minimierung einer Fehlerfunktion mit Hilfe verschiedener Variationsmethoden ist die Finite-Elemente-Methode dann in der Lage, das Problem zu lösen. Im Folgenden wird die grundlegende Vorgehensweise an den in Abschnitt 2.1 beschriebenen Modellproblemen illustriert und in Abbildung 4 zusammengefasst dargestellt. [4, 8, 13, 16]

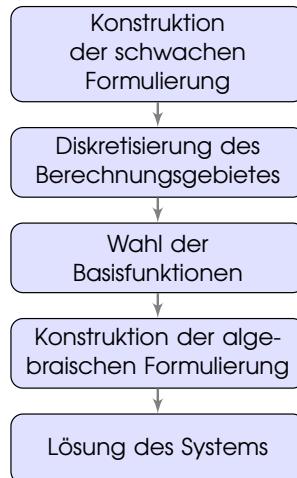


Abbildung 4: Die Abbildung zeigt das schematische Vorgehen bei der Konstruktion der Finite-Elemente-Methode. Das Vorgehen kann prinzipiell auf jedes partielle Differentialgleichungssystem angewendet werden, um einen Algorithmus zu generieren, der dieses unter Verwendung eines Computers numerisch approximiert.

3.1.1 Konstruktion der schwachen Formulierung

Wie in Abschnitt 2.1.2 erwähnt, ist es bereits in der Theorie sinnvoll, die Lösungen von partiellen Differentialgleichungen in einer abgeschwächten Form zu verstehen, da bereits nicht stetige Quellterme die Forderungen an eine klassische Lösung verletzen [16, S. 46]. Demzufolge scheint eine klassische Formulierung von partiellen Differentialgleichungen für die numerische Mathematik ungeeignet. Für die Definition

einer schwachen Lösung beziehungsweise schwachen Formulierung verallgemeinert man die Ableitung einer Funktion mithilfe von Distributionen [16, S. 46 ff]. In den Sobolevräumen werden dann die integrierbaren Funktionen zusammengefasst, die im Sinne einer Distribution wieder eine Ableitung in einem integrierbaren Raum besitzen [16, S. 54 ff].

Auf die Poisson-Gleichung wurde dieses Verfahren in Definition 2.3 bereits angewendet, sodass deren schwache Formulierung hier nicht noch einmal wiederholt werden soll. Allerdings wurde im Abschnitt 2.1.3 bereits erwähnt, dass die für die Finite-Elemente-Methode benötigte schwache Formulierung der Wellengleichung durch die separate Diskretisierung der Zeit erlangt wird [1, 8]. Es sei angemerkt, dass dies nicht die einzige Möglichkeit beschreibt, die Zeit in die Finite-Elemente-Methode einzubauen. Auch diese ließe sich theoretisch gesehen wie die räumlichen Dimensionen durch die Finite-Elemente-Methode behandeln. Dennoch erweist sich dieses Vorgehen der separaten Zeitdiskretisierung als sehr effizient, sowohl bei der numerischen Behandlung als auch bei der Implementierung [8]. Eine detailliertere Betrachtung der Theorie ist dabei in [16, S. 211 ff] und [8, S. 653 ff] nachzulesen.

Wellengleichung

Das Vorkommen der zweiten zeitlichen Ableitung in der Wellengleichung ermöglicht die Anwendung des symplektischen Euler-Verfahrens. Diese Methode erhält, abgesehen von kleinen Schwankungen, die Gesamtenergie eines Systems und führt damit nicht, wie beim expliziten Euler-Verfahren, zur Divergenz der Lösungen. Alternativ ließen sich auch andere Zeitschrittverfahren verwenden.

Zunächst überführt man die Wellengleichung durch die Einführung einer neuen Variable in ein System partieller Differentialgleichungen, in der die zeitliche Ableitung nur in erster Ordnung vorkommt. Es seien also $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ ein Berechnungsgebiet und $([\Omega], f, \varphi_0, \vartheta_0, \varphi^{(D)}, \varphi^{(N)}, \vartheta)$ ein Wellenproblem. In diesem Falle gilt die folgende Äquivalenz für $\vartheta := \partial_t \varphi$.

$$\partial_t^2 \varphi - \Delta \varphi = f \quad \iff \quad \begin{pmatrix} \partial_t \varphi \\ \partial_t \vartheta \end{pmatrix} = \begin{pmatrix} \vartheta \\ \Delta \varphi + f \end{pmatrix}$$

Des Weiteren gelten die folgenden für die Dirichlet-Randbedingungen wichtigen Zusammenhänge für alle $t \in T := [0, \infty)$.

$$\varphi(\cdot, 0) = \varphi_0, \quad \vartheta(\cdot, 0) = \vartheta_0, \quad \vartheta(\cdot, t)|_{\partial\Omega_D} = \partial_t \varphi^{(D)}(\cdot, t)|_{\partial\Omega_D}$$

Es soll nun ein kleiner Zeitschritt $dt \in (0, \infty)$ gewählt werden, der das Zeitintervall T durch $[T] := \{k \cdot dt \mid k \in \mathbb{N}_0\}$ diskretisiert. Weiterhin bezeichne eine natürliche Zahl $n \in \mathbb{N}$ als Index der zeitabhängigen Funktionen $f, \varphi^{(D)}, \varphi^{(N)}, \varphi$ und ϑ deren Diskretisierung zum Zeitpunkt $n \cdot dt$. Dieses Schema soll im Folgenden am Beispiel von φ demonstriert werden.

$$\varphi \longleftrightarrow (\varphi_n)_{n \in \mathbb{N}_0}, \quad \varphi(\cdot, ndt) \longleftrightarrow \varphi_n$$

Für die Dirichlet-Randbedingungen von ϑ_n soll auch hier ein einfaches Euler-Verfahren verwendet werden. Da die Randbedingungen von vornherein bekannt sind, wäre hier auch die Wahl eines analytischen Verfahrens möglich.

$$\vartheta(\cdot, ndt)|_{\partial\Omega_D} = \partial_t \varphi^{(D)}(\cdot, ndt) \Big|_{\partial\Omega_D} \quad \longleftrightarrow \quad \vartheta_n|_{\partial\Omega_D} = \frac{\varphi_n^{(D)} - \varphi_{n-1}^{(D)}}{dt} \Big|_{\partial\Omega_D}$$

Für ein solches System kann nun das symplektische Euler-Verfahren explizit notiert werden. Die Idee besteht darin, nicht alle Terme auf der linken Seite der Gleichung gleichzeitig zu berechnen. In diesem Falle wird die Funktion ϑ_{n+1} als Erstes berechnet, um die Berechnung von φ_{n+1} zu ermöglichen.

$$\begin{pmatrix} \partial_t \varphi \\ \partial_t \vartheta \end{pmatrix} = \begin{pmatrix} \vartheta \\ \Delta \varphi + f \end{pmatrix} \quad \longleftrightarrow \quad \frac{1}{dt} \left[\begin{pmatrix} \varphi_{n+1} \\ \vartheta_{n+1} \end{pmatrix} - \begin{pmatrix} \varphi_n \\ \vartheta_n \end{pmatrix} \right] = \begin{pmatrix} \vartheta_{n+1} \\ \Delta \varphi_n + f_n \end{pmatrix}$$

Umgestellt nach φ_{n+1} und ϑ_{n+1} erhält man nun die explizite Berechnungsformel eines Zeitschrittes.

$$\begin{pmatrix} \varphi_{n+1} \\ \vartheta_{n+1} \end{pmatrix} = \begin{pmatrix} \varphi_n \\ \vartheta_n \end{pmatrix} + dt \begin{pmatrix} \vartheta_{n+1} \\ \Delta \varphi_n + f_n \end{pmatrix}$$

Die Konstruktion der schwachen Formulierung kann nun durch dieselbe Vorgehensweise wie bei der Poisson-Gleichung erreicht werden. Zunächst werden wieder durch Hilfsvariablen die Randbedingungen eingearbeitet.

$$\begin{aligned} \varphi_{n+1}^\circ &:= \varphi_{n+1} - \varphi_{n+1}^{(D)} & \varphi_{n+1}^\circ &\in H_D^1(\Omega) \\ \vartheta_{n+1}^\circ &:= \vartheta_{n+1} - \vartheta_{n+1}|_{\partial\Omega_D} & \vartheta_{n+1}^\circ &\in H_D^1(\Omega) \end{aligned}$$

Die erste Gleichung des Systems für φ_{n+1} ändert sich im Prinzip nicht, da keine räumlichen Ableitungen enthalten sind. Das Einsetzen der Hilfsvariablen ergibt die folgende Gleichung.

$$\varphi_{n+1}^\circ = \varphi_n^\circ + dt \vartheta_{n+1}^\circ$$

Die Umformulierung der Gleichung für ϑ_{n+1} ergibt wieder einen länglichen Ausdruck für alle $\xi \in H_D^1(\Omega)$.

$$\begin{aligned} \int_\Omega \vartheta_{n+1}^\circ \xi \, d\lambda &= \int_\Omega \vartheta_n \xi \, d\lambda - \frac{1}{dt} \left[\int_\Omega \varphi_{n+1}^{(D)} \xi \, d\lambda - \int_\Omega \varphi_n^{(D)} \xi \, d\lambda \right] \\ &\quad + dt \left[\int_\Omega f_n \xi \, d\lambda + \int_{\partial\Omega_N} \varphi_n^{(N)} \xi \, d\sigma - \int_\Omega \langle \nabla \varphi_n, \nabla \xi \rangle \, d\lambda \right] \end{aligned}$$

Bei der Implementierung der Wellengleichungen werden später der Übersicht halber immer natürliche Neumann- und Dirichlet-Randbedingungen mit $f = 0$ gewählt werden, sodass sich die schwache Formulierung in diesem Fall wie folgt ergibt.

$$\int_\Omega \vartheta_{n+1}^\circ \xi \, d\lambda = \int_\Omega \vartheta_n^\circ \xi \, d\lambda - dt \int_\Omega \langle \nabla \varphi_n^\circ, \nabla \xi \rangle \, d\lambda$$

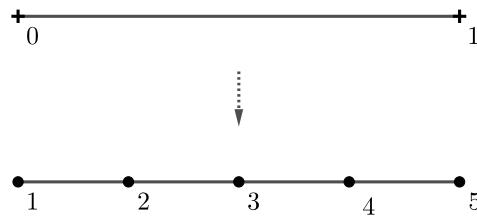


Abbildung 5: Die Abbildung zeigt den Übergang von einem eindimensionalen Berechnungsgebiet in ein diskretisiertes Gebiet. Anstatt eines Intervalls wird das diskretisierte Gebiet nur noch durch Eckpunkte und deren Verbindungsstücke beschrieben.

3.1.2 Diskretisierung des Berechnungsgebietes

Aufgrund der Funktionsweise eines Computers ist man bei numerischen Simulationen gezwungen, kontinuierliche Probleme auf die eine oder andere Art und Weise zu diskretisieren. In der Finite-Elemente-Methode findet diese Diskretisierung durch die Einführung von finiten Elementen statt. Das Berechnungsgebiet Ω und dessen Rand $\partial\Omega$ werden in ein äquivalentes System von mehreren finiten Elementen überführt. Dabei wird die Anzahl, der Typ und die Größe dieser Elemente durch den Modellieerer festgelegt. Für zwei Raumdimensionen werden zumeist Dreiecke und Vierecke gewählt. Die Abbildungen 5 und 6 veranschaulichen dieses Vorgehen im ein- und zweidimensionalen Fall anhand schematischer Beispiele. Das weitere Vorgehen der nächsten Paragraphen wird am Beispiel des eindimensionalen Berechnungsgebiet aus Abbildung 5 gezeigt. [1, 4, 8, 13]

Sowohl im ein- und zweidimensionalen Fall ist zuallererst die Wahl von Eckpunkten (engl.: *vertex*), die den Rand und das innere des Berechnungsgebietes näherungsweise beschreiben, nötig. Im Anschluss daran ist die Wahl der finiten Elemente zu treffen. Im eindimensionalen Beispiel aus Abbildung 5 muss es sich dabei um Kanten handeln, die benachbarte Eckpunkte verbinden. Im zweidimensionalen Fall sollen in dieser Arbeit als finite Elemente nur Dreiecke gewählt werden. Andere Arten von

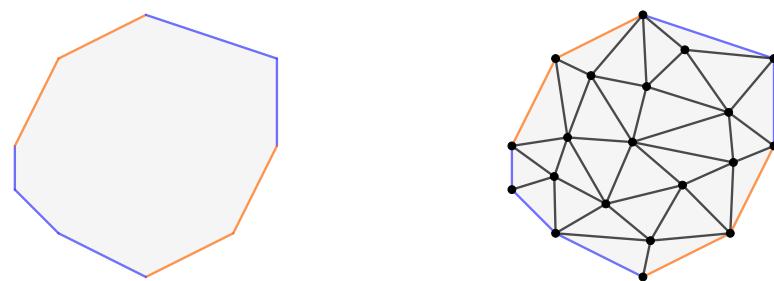


Abbildung 6: Die rechte Abbildung zeigt ein Beispiel für die Diskretisierung des polygonalen Berechnungsgebietes der linken Abbildung durch die Verwendung von Dreiecken. Die Dreiecke werden hierbei auch als finite Elemente bezeichnet.

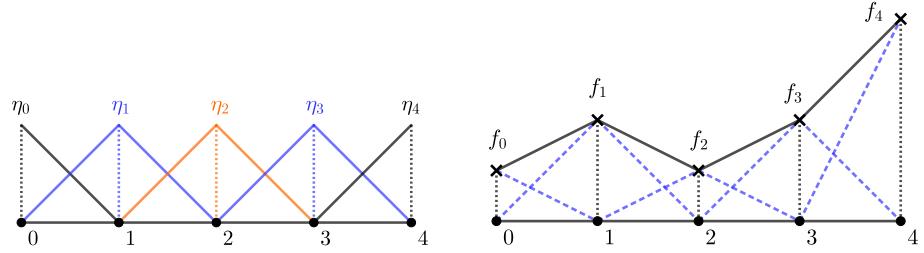


Abbildung 7: Die linke Abbildung zeigt die eindimensionalen Hutfunktionen η_i für jeden Eckpunkt $i \in \{0, 1, 2, 3, 4\}$. Für den diskretisierten Raum von Funktionen stellen diese gerade die Basisfunktionen dar und beschreiben eine lineare Interpolation zwischen den Eckpunkten, wie sie im rechten Bild zu sehen ist. Die durch die Koordinaten f_i beschriebene Funktion $f: [0, 1] \rightarrow \mathbb{R}$ errechnet sich zu $f(x) := \sum_{i=0}^4 f_i \eta_i(x)$.

finiten Elementen können vollkommen analog zu diesen eingeführt werden. Auch die Dreiecke definieren auf der Menge der Eckpunkte eine Art Nachbarschaft. Wichtig dabei ist, dass sich nur die Kanten der Dreiecke schneiden dürfen, nicht aber die Dreiecke selbst, da sonst ein Teil des Berechnungsgebietes durch mehr als ein Dreieck beschrieben werden würde.

Für den weiteren Verlauf beschreibe V die endliche nichtleere Menge der gewählten Eckpunkte in einem Berechnungsgebiet. Um einfacher mit der Diskretisierung von Randwerten und -integralen umzugehen, soll zudem V° die Menge aller Eckpunkte, die nicht auf dem Dirichlet-Rand $\partial\Omega_D$ liegen, V_D die Menge aller Eckpunkte, die auf dem Dirichlet-Rand $\partial\Omega_D$ liegen, und V_N die Menge aller Eckpunkte, die auf dem Neumann-Rand $\partial\Omega_N$ liegen, bezeichnen. In diesem Falle gelten die folgenden Beziehungen.

$$V = V^\circ \cup V_D, \quad V^\circ \cap V_D = \emptyset, \quad V_N \subset V^\circ$$

Die Menge der Dreiecke beziehungsweise Kanten soll hier nicht weiter beschrieben werden, weil diese auch durch die Wahl der Basisfunktionen im nächsten Abschnitt gegeben ist.

3.1.3 Wahl der Basisfunktionen

Neben der Diskretisierung des Berechnungsgebietes $[\Omega]$ ist auch die Diskretisierung der Funktionenräume $H^1(\Omega)$ und $H_D^1(\Omega)$ nötig. Dies erreicht man durch die Wahl endlich vieler Basisfunktionen. Im Idealfall sollten diese einen kleinen Träger aufweisen, um die spätere Berechnung effizienter zu gestalten. Gerade für partielle Differentialgleichungen zweiter Ordnung ist die Wahl von stückweise linearen Basisfunktionen ausreichend, da sie, wie auch deren schwachen Lösungen, dem Raum $H^1(\Omega)$ angehören. Die genannten Basisfunktionen werden aufgrund ihrer Form auch Hutfunktionen genannt. [1, 4, 8, 13, 16]

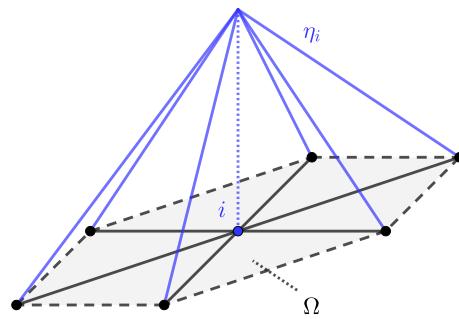


Abbildung 8: Die Abbildung zeigt die typische Wahl einer Basisfunktion η_i über dem i . Vertex des diskretisierten Berechnungsgebietes Ω . Aufgrund ihrer Form wird sie auch Hutfunktion genannt. Sie ist stückweise linear und identisch zu Null auf nicht benachbarten Dreiecken. Sie entspricht einer linearer Interpolation zwischen den Eckpunkten.

Eindimensionale Hutfunktionen

Das Prinzip der Hutfunktionen soll im Folgenden am Beispiel des aus Abbildung 5 bekannten eindimensionalen Berechnungsgebietes genauer erklärt werden. Abbildung 7 stellt die Hutfunktionen und eine aus ihnen berechnete Beispiefunktion über dem Berechnungsgebiet schematisch dar. Es sei nun eine Kante $[x_i, x_j] \subset \mathbb{R}$ zwischen den aufeinanderfolgenden Eckpunkten $i \in \{0, 1, 2, 3\}$ und $j = i + 1$ gegeben. Über dieser Kante lassen sich durch eine Parametrisierung γ die stückweisen linearen Basisfunktionen leicht definieren. Der Definitionsbereich der Basisfunktionen ist jedoch auf die gegebene Kante eingeschränkt, um eine leichtere Handhabung zu gestatten. Die Gleichungen müssen für jede Kante formuliert werden.

$$\gamma: [0, 1] \rightarrow [x_i, x_j], \quad \gamma(s) := x_i(1 - s) + x_js$$

$$\eta_i, \eta_j: [x_i, x_j] \rightarrow \mathbb{R}, \quad \eta_i \circ \gamma(s) := 1 - s, \quad \eta_j \circ \gamma(s) := s$$

Bei der Parametrisierung γ handelt es sich um eine bijektive Funktion. Durch deren Invertierung ist eine explizite Formulierung der Basisfunktionen für alle $x \in [x_i, x_j]$ möglich.

$$\eta_i(x) = \frac{x_j - x}{x_j - x_i}, \quad \eta_j(x) = \frac{x - x_i}{x_j - x_i}$$

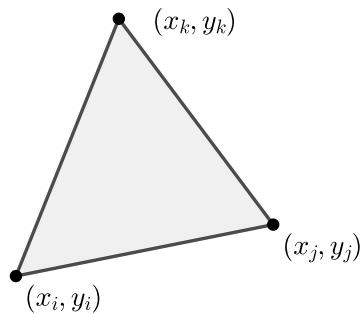
Die Menge der Hutfunktionen bildet eine Basis des diskretisierten Funktionenraumes. Da es sich in diesem Beispiel um fünf Basisfunktionen handelt, besitzt der Funktionenraum ebenfalls die Dimension fünf. Jede Funktion $f: [0, 1] \rightarrow \mathbb{R}$ dieses Raums lässt sich demnach als eine Linearkombination der fünf Basisfunktionen darstellen. Es seien hierfür die Koordinaten $f_i \in \mathbb{R}$ für alle $i \in \{0, 1, 2, 3, 4\}$ gegeben.

$$f = \sum_{i=0}^4 f_i \eta_i$$

Die Bedeutung der Linearkombination für die Interpolation wird im rechten Bild von Abbildung 7 gezeigt.

Zweidimensionale Hutfunktionen auf dem Dreieck

Abbildung 8 stellt eine zweidimensionale Hutfunktion über einem Eckpunkt des diskretisierten Berechnungsgebietes dar. Der Träger der Funktion ist auf benachbarte Dreiecke beschränkt. Außerhalb dieser Dreiecke ist die Hutfunktion identisch zu Null. Für die folgenden Formeln und Berechnungsschritte wurde das Vorgehen aus [1] gewählt.



Für den zweidimensionalen Fall betrachte man ein Dreieck, welches durch die Eckpunkte (x_i, y_i) , (x_j, y_j) und (x_k, y_k) aus der Menge \mathbb{R}^2 gegeben ist. Die Menge aller Punkte des Dreiecks sei definiert als Δ . Um die Hutfunktionen auf eine einfache Weise zu definieren, konstruiert man auch für dieses Dreieck zunächst eine Parametrisierung, die auf den baryzentrischen Koordinaten basiert.

$$M := \{(u, v) \in [0, 1]^2 \mid u + v \leq 1\}$$

M dient hier als der Definitionsbereich dieser Parametrisierung. Die Hutfunktionen seien hier mit η_i , η_j und η_k bezeichnet. Zu beachten ist dabei, dass der Definitionsbereich der Basisfunktionen wieder auf das zugrundeliegende Dreieck eingeschränkt ist. Die Gleichungen müssen also auch hier für jedes Dreieck des diskretisierten Berechnungsgebietes formuliert werden.

$$\gamma: M \rightarrow \Delta, \quad \gamma(u, v) := (1 - u - v) \begin{pmatrix} x_i \\ y_i \end{pmatrix} + u \begin{pmatrix} x_j \\ y_j \end{pmatrix} + v \begin{pmatrix} x_k \\ y_k \end{pmatrix}$$

$$\eta_i, \eta_j, \eta_k: \Delta \rightarrow \mathbb{R}$$

$$\eta_i \circ \gamma(u, v) := 1 - u - v, \quad \eta_j \circ \gamma(u, v) := u, \quad \eta_k \circ \gamma(u, v) := v$$

Um nun die Basisfunktionen in expliziter Form zu erhalten, muss die bijektive Parametrisierung γ invertiert werden. Dies geschieht durch ein lineares Gleichungssystem, welches auf dem Papier durch ein analytisches Verfahren leicht aufgelöst werden kann. Die genauen Rechnungen sollen hier aus Platzgründen nicht vorgeführt werden. Man erhält damit für alle $(x, y) \in \Delta$ die folgenden Ausdrücke.

$$\eta_i(x, y) = \frac{1}{\alpha} \begin{vmatrix} 1 & x & y \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{vmatrix} \quad \eta_j(x, y) = \frac{1}{\alpha} \begin{vmatrix} 1 & x & y \\ 1 & x_k & y_k \\ 1 & x_i & y_i \end{vmatrix}$$

$$\eta_k(x, y) = \frac{1}{\alpha} \begin{vmatrix} 1 & x & y \\ 1 & x_i & y_i \\ 1 & x_j & y_j \end{vmatrix} \quad \alpha := \begin{vmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{vmatrix}$$

Wie auch im eindimensionalen Fall wird der diskretisierte Funktionenraum S durch die Menge der Linearkombination der Basisfunktionen beschrieben. Es sei dabei V wieder die Menge aller Eckpunkte des diskretisierten Berechnungsgebietes.

$$S := \text{span} \{ \eta_i \mid i \in V \} \subset H^1(\Omega), \quad S_D := S \cap H_D^1(\Omega)$$

3.1.4 Konstruktion der algebraischen Formulierung

Nachdem sowohl das Berechnungsgebiet als auch die Funktionenräume diskretisiert wurden, ist es nun möglich, darauf aufbauend aus den schwachen Formulierungen der partiellen Differentialgleichungen ein System algebraischer Gleichungen zu erlangen. Grundsätzlich ersetzt das Vorgehen alle Funktionen aus den Räumen $H_D^1(\Omega)$ und $H^1(\Omega)$ durch die stückweisen linearen Funktionen der Räume S_D und S . Das Verfahren soll zunächst anhand des Poisson-Problems demonstriert werden. Das resultierende lineare Gleichungssystem lässt sich dann analog auch für das Wellen- und Wärmeleitungsproblem herleiten.

Poisson-Problem

Es sei wieder $([\Omega], f, u^{(D)}, u^{(N)}, u)$ ein Poisson-Problem. Nach Definition 2.3 ist die schwache Formulierung des Poisson-Problems durch die folgende Gleichung für alle $\varphi \in H_D^1(\Omega)$ gegeben.

$$\int_{\Omega} \langle \nabla v, \nabla \varphi \rangle d\lambda = \int_{\Omega} f \varphi d\lambda + \int_{\partial\Omega_N} u^{(N)} \varphi d\sigma - \int_{\Omega} \langle \nabla u^{(D)}, \nabla \varphi \rangle d\lambda$$

Weiterhin sei die Menge der Eckpunkte des diskretisierten Berechnungsgebietes durch V und deren zugehörige Basisfunktionen durch die Menge $B := \{\eta_i : \Omega \rightarrow \mathbb{R} \mid i \in V\}$ gegeben. Um die Notation etwas leichter zu gestalten, werden die folgenden natürlichen Zahlen eingeführt.

$$n := |V|, \quad m := |V^\circ|, \quad p := |V_D|, \quad q := |V_N|$$

Alle Funktionen des Poisson-Problems sollen nun durch den diskreten Raum der Funktionen beschrieben werden. Hierfür wählt man die folgenden Vektoren $[f], [u] \in \mathbb{R}^n$, $[u^{(D)}] \in \mathbb{R}^p$, $[u^{(N)}] \in \mathbb{R}^q$, $[v] \in \mathbb{R}^m$. Diese Vektoren beschreiben die Koordinaten bezüglich der gewählten Basisfunktionen.

$$\begin{aligned} f &= \sum_{i \in V} [f]_i \eta_i & u^{(D)} &= \sum_{i \in V_D} [u^{(D)}]_i \eta_i \\ v &= \sum_{i \in V^\circ} [v]_i \eta_i & u^{(N)} &= \sum_{i \in V_N} [u^{(N)}]_i \eta_i|_{\partial\Omega_N} \end{aligned}$$

$$u = \sum_{i \in V} [u]_i \eta_i = \underbrace{\sum_{i \in V^\circ} [v]_i \eta_i}_{=v} + \underbrace{\sum_{i \in V_D} [u^{(D)}]_i \eta_i}_{=u^{(D)}} = v + u^{(D)}$$

Diese verschiedenen Ausdrücke werden nun in das Poisson-Problem eingesetzt. In der schwachen Formulierung des Poisson-Problems kann das $\varphi \in H_D^1(\Omega)$ direkt durch alle Basisfunktionen ersetzt werden. Man erhält damit für alle $i \in V^\circ$ die folgende Gleichung.

$$\begin{aligned} \sum_{j \in V^\circ} u_j \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda &= \sum_{j \in V} f_j \int_{\Omega} \eta_i \eta_j d\lambda + \sum_{j \in V_N} u_j^{(N)} \int_{\partial \Omega_N} \eta_i \eta_j d\sigma \\ &\quad - \sum_{j \in V_D} u_j^{(D)} \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda \end{aligned}$$

Fasst man dies zu einem System von Gleichungen zusammen, so entsteht ein lineares Gleichungssystem. Für eine einfachere Notation führt man aus diesem Grund die Matrizen $[\Delta] \in \mathbb{R}^{m \times m}$, $[M] \in \mathbb{R}^{m \times n}$, $[N] \in \mathbb{R}^{m \times q}$ und $[D] \in \mathbb{R}^{m \times p}$ ein.

$$\begin{aligned} [\Delta]_{ij} &:= \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda & [M]_{ij} &:= \int_{\Omega} \eta_i \eta_j d\lambda \\ [D]_{ij} &:= \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda & [N]_{ij} &:= \int_{\partial \Omega_N} \eta_i \eta_j d\sigma \end{aligned}$$

Die Matrix $[\Delta]$ wird auch »Stiffness Matrix« und $[M]$ auch »Mass Matrix« genannt. In der einfacheren Matrix-Vektor-Notation lässt sich das System linearer Gleichungen nun wie folgt schreiben.

$$[\Delta][v] = [M][f] + [N] \left[u^{(N)} \right] - [\Delta] \left[u^{(D)} \right]$$

Bei $[\Delta]$ handelt es sich um eine symmetrische, positiv-definite Matrix [1]. Für das System linearer Gleichungen existiert damit eine eindeutige Lösung [1, 16].

Wellenproblem

Für die Wellengleichung wird ein analoges Verfahren verwendet. Es sollen jedoch die Dirichlet-Randpunkte von den restlichen Punkten separiert werden. Es sei also $([\Omega], f, u_0, u_1, u_D, u_N, u)$ ein Wellenproblem. Dann lässt sich ein Matrix-Vektor-Produkt in der folgenden Form schreiben.

$$[M][\vartheta_n] =: \underbrace{[M]^\circ [\vartheta_n]^\circ}_{\text{innere Eckpunkte}} + \underbrace{[M]^{(D)} [\vartheta_n]^{(D)}}_{\text{Dirichlet-Eckpunkte}}$$

Dieses Verfahren wird nun auf die schwache Formulierung der zeit-diskretisierten Wellengleichung angewendet, wodurch man das folgende lineare Gleichungssystem erhält. Auffallend ist, dass in der Finite-Elemente-Formulierung der Wellengleichung

dieselben Matrizen, wie in der Finite-Elemente-Formulierung der Poisson-Gleichung vorkommen.

$$\begin{aligned} [M]^\circ [\vartheta_{n+1}]^\circ &= [M]^\circ [\vartheta_n]^\circ + [M]^{(D)} [\vartheta_n]^{(D)} \\ &\quad - \frac{1}{dt} \left([M]^{(D)} [\varphi_{n+1}]^{(D)} - [M]^{(D)} [\varphi_n]^{(D)} \right) \\ &\quad + dt \left([M]^\circ [f_n]^\circ + [M]^{(D)} [f_n]^{(D)} + [N] [\varphi_n]^{(N)} \right. \\ &\quad \left. - [\Delta]^\circ [\varphi_n]^\circ - [\Delta]^{(D)} [\varphi_n]^{(D)} \right) \end{aligned}$$

Finite-Elemente-Matrizen

Ein letzter Schritt für die Formulierung des linearen Gleichungssystems kümmert sich nun darum, die komplexen Systemmatrizen aus den einfachen Matrizen der finiten Elemente zu konstruieren. Es sei \mathcal{T} die Menge der finiten Elemente, die das Berechnungsgebiet diskretisieren. Das Integral über Ω lässt sich für eine beliebige Funktion damit als Summe der Integrale über die finiten Elemente darstellen.

$$\int_{\Omega} \eta_i \eta_j d\lambda = \sum_{E \in \mathcal{T}} \int_E \eta_i \eta_j d\lambda, \quad \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda = \sum_{E \in \mathcal{T}} \int_E \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda$$

Durch die kleinen Träger der Basisfunktionen sind die beschriebenen Integrale häufig Null. Wählt man demnach zwei Basisfunktionen η_i und η_j , deren Eckpunkte i und j nicht durch die Kante eines finiten Elements verbunden sind, so ergibt sich das Folgende für alle $E \in \mathcal{T}$.

$$\int_E \eta_i \eta_j d\lambda = \int_{\Omega} \eta_i \eta_j d\lambda = 0, \quad \int_E \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda = \int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda = 0$$

Ein ähnliches Vorgehen kann auch für die Integrale über dem Rand $\partial\Omega_N$ gewählt werden. Die komplexen Systemmatrizen ergeben sich damit als eine Summe von einfach zu beschreibenden Finite-Elemente-Matrizen.

Eindimensionales Beispiel des Poisson-Problems

Das lineare Gleichungssystem soll einmal für das eindimensionale Beispiel aus Abbildung 5 notiert werden. Dabei sollen an den Randpunkten Dirichlet-Randbedingungen herrschen. Die Neumann-Randbedingungen werden also für dieses Beispiel vernachlässigt. Als Erstes werden die einfachen Matrizen jeder einzelnen Kante $[x_i, x_j]$ mit $i \in \{0, 1, 2, 3\}$ und $j = i + 1$ konstruiert. Die Länge aller Kanten ist gleich und soll als $h := x_j - x_i$ bezeichnet werden.

$$\begin{aligned} \left(\int_{[x_i, x_j]} \eta_p \eta_q d\lambda \right)_{p,q \in \{i,j\}} &= \frac{h}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \\ \left(\int_{[x_i, x_j]} \langle \nabla \eta_p, \nabla \eta_q \rangle d\lambda \right)_{p,q \in \{i,j\}} &= \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \end{aligned}$$

Aus der Addition der einzelnen Finite-Elemente-Matrizen ergeben sich dann die kompletten System-Matrizen, auch »Mass Matrix« und »Stiffness Matrix« genannt. In den folgenden Matrizen wurden Einträge mit dem Wert Null nicht explizit notiert.

$$\begin{aligned} \left(\int_{\Omega} \eta_i \eta_j d\lambda \right)_{i \in V^{\circ}, j \in V} &= \frac{h}{6} \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \end{pmatrix} \\ \left(\int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda \right)_{i,j \in V^{\circ}} &= \frac{1}{h} \begin{pmatrix} 2 & -1 & \\ -1 & 2 & -1 \\ & -1 & 2 \end{pmatrix} \\ \left(\int_{\Omega} \langle \nabla \eta_i, \nabla \eta_j \rangle d\lambda \right)_{i \in V^{\circ}, j \in V_D} &= \frac{1}{h} \begin{pmatrix} -1 & 0 \\ 0 & 0 \\ 0 & -1 \end{pmatrix} \end{aligned}$$

Diese setzt man nun in die Gesamtgleichung ein und erhält das folgende Gleichungssystem, welches nach $[v]$ aufgelöst werden soll. Die »Stiffness Matrix« ist symmetrisch und positiv definit. Das System besitzt damit eine eindeutige Lösung.

$$\frac{1}{h} \begin{pmatrix} 2 & -1 & \\ -1 & 2 & -1 \\ & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \frac{h}{6} \begin{pmatrix} 1 & 4 & 1 & & \\ 1 & 4 & 1 & 1 & \\ 1 & 4 & 1 & & \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} - \frac{1}{h} \begin{pmatrix} -1 & 0 \\ 0 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} u_0^{(D)} \\ u_4^{(D)} \end{pmatrix}$$

Konstruktion der Matrizen im Zweidimensionalen

Für die Berechnung der System-Matrizen eines zweidimensionalen Berechnungsgebiets ist ein analoges Vorgehen möglich. Die Integrale des vorigen Abschnittes müssen auch hier über den finiten Elementen ausgewertet und zu einer Systemmatrix zusammengesetzt werden. Für die Implementierung des Algorithmus reicht es demnach, die Finite-Elemente-Matrizen anzugeben. Die Berechnung dieser soll aufgrund des Umfangs ihrer Herleitungen nicht angegeben werden.

$$\begin{aligned} \left(\int_{\Delta} \eta_p \eta_q d\lambda \right)_{p,q \in \{i,j,k\}} &= \frac{\lambda(\Delta)}{12} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \\ \left(\int_{\Delta} \langle \nabla \eta_p, \nabla \eta_q \rangle d\lambda \right)_{p,q \in \{i,j,k\}} &= \frac{1}{4\lambda(\Delta)} D^T D \\ D := & \begin{pmatrix} x_k - x_j & x_i - x_k & x_j - x_i \\ y_k - y_j & y_i - y_k & y_j - y_i \end{pmatrix} \\ \left(\int_E \eta_p \eta_q d\sigma \right)_{p,q \in \{i,j\}} &= \frac{\sigma(E)}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \end{aligned}$$

3.1.5 Lösung des algebraischen Systems

Die erhaltenen linearen Gleichungssysteme sind äquivalent zu der schwachen Formulierung des diskretisierten Problems und werden durch dünnbesetzte Matrizen charakterisiert. Zudem sind die hier konstruierten inneren Systemmatrizen symmetrisch und positiv definit. Daraus folgt, dass für die linearen Gleichungssysteme eine eindeutige Lösung existiert [1, 16]. Für die Lösung dieser Systeme ist eine Vielzahl an unterschiedlichen Methoden, wie zum Beispiel das CG-Verfahren, welches in Abschnitt 3.2.2 erläutert wird, dokumentiert.

3.2 Numerische Methoden dünnbesetzter Matrizen

Dünnbesetzte Matrizen (engl.: *sparse matrix*) treten in einem beträchtlichen Anteil numerischer Disziplinen auf und als ein Resultat sind Methoden, um diese effizient zu manipulieren, häufig ein kritischer Punkt der Performance vieler Anwendungen. Vor allem die Matrix-Vektor-Multiplikation dünnbesetzter Systeme repräsentiert zumeist die dominanten Kosten vieler iterativer Lösungsmethoden linearer Systeme großen Ausmaßes [2]. Gerade bei der Finite-Elemente-Methode werden Mass und Stiffness Matrix im Computer durch ein Sparse-Matrix-Format dargestellt [1, 2, 8]. Die Wahl der Speicherformate und Lösungsmethoden wirken sich dabei extrem auf die Performance der Anwendung aus [2, 3].

3.2.1 Datenstrukturen und das CSR-Format

Eine Sparse-Matrix $M \in \mathbb{R}^{n \times n}$ mit $n \in \mathbb{N}$, deren Anzahl von Elementen ungleich Null der Größenordnung n entspricht, kann durch das Allozieren aller n^2 Speicherplätze nicht effizient dargestellt werden. Häufig stellt dieses Unterfangen sogar eine physikalische Unmöglichkeit dar, weil ab einer bestimmten Größe von n der Speicherplatz für die n^2 Elemente nicht einmal existiert. Selbst wenn man den Speicher allozieren könnte, so wäre eine Iteration über alle Elemente, eine häufige Operation bei der Verwendung von Matrizen, reine Zeitverschwendug, da ein Großteil dieser Elemente Null wäre. Offensichtlich müssen also gewisse indizierte Speicherschemen verwendet werden, um die Natur einer dünnbesetzten Matrix effizient zu beschreiben. [12, S. 78]

Typische Datenstrukturen für die Verwendung dünnbesetzter Matrizen werden durch die Formate COO (engl.: *coordinate list*), CSR (engl.: *compressed sparse row*) und ELLPACK (ELL) dargestellt [2, 3]. Allerdings existieren wesentlich mehr Formate, die im Rahmen dieser Arbeit nicht erwähnt werden können. Untersuchungen über die Implementierung der genannten Formate auf der GPU sind in [2] und [3] zu finden. Jedes Format besitzt Vor- und Nachteile. Bis auf das COO-Format scheint die Effizienz zudem vom genauen Muster der dünnbesetzten Matrix abzuhängen. Es stellte sich heraus, dass das CSR-Format eine sehr gute durchschnittliche Performance

ermöglicht. Aus diesem Grund beschränkt sich diese Arbeit auf die Beschreibung und Implementierung dessen. Eine genauere Erläuterung des ELL-Formates ist im Anhang zu finden.

Für eine Matrix $M \in \mathbb{R}^{n \times n}$ mit $k \in \mathbb{N}, k \leq n^2$ Einträgen ungleich Null speichert das CSR-Format diese in einem Tupel $[M]_{\text{CSR}}$ dreier Vektoren.

$$[M]_{\text{CSR}} := (V, R, C), \quad V \in \mathbb{R}^k, \quad R \in \mathbb{N}_0^{n+1}, \quad C \in \mathbb{N}_0^k$$

In V werden dabei alle Werte ungleich Null gespeichert, indem Zeile für Zeile alle Spalten von links nach rechts ausgelesen werden. Dies bedeutet, dass alle Werte a_{ij} von M mit $(i, j) \in \mathbb{N}^2$ und $i, j \leq n$, die ungleich Null sind, anhand von $m(i-1, j-1)$ geordnet werden.

$$m: \{p \in \mathbb{N}_0 \mid p < n\}^2 \rightarrow \mathbb{N}_0, \quad m(i, j) := n \cdot i + j$$

Für jedes $i \in \mathbb{N}$ mit $i \leq n+1$ speichert R_i die Anzahl der Werte ungleich Null, die bis zur Zeile i von M vorkommen. R_0 wird dabei immer auf Null gesetzt. Demnach kann R als kumulative Verteilungsfunktion der Einträge ungleich Null über die Zeilen verstanden werden. In C werden die Spaltenindizes eines jeden Wertes aus V gespeichert. Dieses Vorgehen soll an dem folgenden Beispiel gezeigt werden [19].

$$M = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}, \quad [M]_{\text{CSR}} = \left(\begin{pmatrix} 3 \\ 22 \\ 17 \\ 7 \\ 5 \\ 1 \\ 6 \\ 14 \\ 8 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 3 \\ 0 \\ 5 \\ 6 \\ 1 \\ 8 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 4 \\ 0 \\ 1 \\ 3 \\ 2 \\ 4 \end{pmatrix} \right)$$

Für Berechnung des Matrix-Vektor-Produktes eines Vektors $x \in \mathbb{R}^n$ mit der Matrix $M = (a_{ij})_{i,j}$, gespeichert durch $[M]_{\text{CSR}}$, wird im einfachsten Fall zunächst zeilenweise gearbeitet. Für jede Zeile $i \in \mathbb{N}$ mit $i \leq n$ muss ein einfaches Skalarprodukt ausgewertet werden.

$$(Mx)_i = \sum_{j=1}^n a_{ij} x_j = \sum_{j=R_i+1}^{R_{i+1}} V_j x_{C_j+1} = ([M]_{\text{CSR}} x)_i$$

Ein analoges Verfahren ist das sogenannten CSC-Format (engl.: *compressed sparse column*). Hier wird die kumulative Verteilungsfunktion im Bezug auf die Spalten und nicht auf die Zeilen gebildet. [2]

3.2.2 Lösung dünnbesetzter Systeme und das CG-Verfahren

Das CG-Verfahren (engl.: *conjugate gradient method*) ist eine iterative Methode, um das folgende lineare Gleichungssystem zu lösen, welches durch eine symmetrische, positiv definite Matrix $A \in \mathbb{R}^{n \times n}$, einen Vektor $b \in \mathbb{R}^n$ und eine eindeutige Lösung $x \in \mathbb{R}^n$ für $n \in \mathbb{N}$ beschrieben wird.

$$Ax = b$$

Eine äquivalente Formulierung ist dabei durch das folgende Optimierungsproblem zusammen mit dessen Residuum gegeben.

$$\min_{x \in \mathbb{R}^n} \varphi(x) := x^T A x - b^T x, \quad \nabla \varphi(x) = Ax - b =: r(x)$$

Das CG-Verfahren konvergiert nach spätestens n Schritten zur korrekten Lösung. Insbesondere nimmt mit jeder Iteration der Gesamtfehler der approximierten Lösung ab, sodass das Verfahren bereits nach einer geringeren Anzahl an Iterationen abgebrochen werden kann.

Grundsätzlich basiert das Verfahren auf der Konstruktion von n zueinander paarweise A -konjugierten Vektoren p_i für $i \in \mathbb{N}$ mit $i \leq n$. Auf der Grundlage dieser Vektoren kann dann in jeder Iteration ein neuer Lösungsvektor konstruiert werden, dessen Gesamtfehler reduziert wurde. Der folgende Algorithmus wurde [10] entnommen und beschreibt das Verfahren zur Berechnung des Lösungsvektors x , sofern ein beliebiger Startwert $x_0 \in \mathbb{R}^n$ gegeben ist. Für eine detailliertere Behandlung des CG-Verfahrens sei hier auf [10] verwiesen. Dort wird auch die Möglichkeit des sogenannten Vorkonditionierens beschrieben, die die Konvergenzgeschwindigkeit des Verfahrens noch erhöht. [10]

In Abschnitt 4.4 wurde das hier beschriebene CG-Verfahren in der Sprache C++ auf der CPU und auf der GPU unter Zuhilfenahme der Sprache CUDA, der Bibliothek »Eigen« und der Bibliothek »Thrust« implementiert [6, 19].

Algorithmus: CG-Verfahren

Given: $A \in \mathbb{R}^{n \times n}$, $x_0 \in \mathbb{R}^n$, $b \in \mathbb{R}^n$

$$r_0 \leftarrow Ax_0 - b$$

$$p_0 \leftarrow -r_0$$

$$k \leftarrow 0$$

while $r_k \neq 0$

$$\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$$

$$x_{k+1} \leftarrow x_k + \alpha_k p_k$$

$$r_{k+1} \leftarrow r_k + \alpha_k A p_k$$

$$\beta_{k+1} \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

$$p_{k+1} \leftarrow -r_{k+1} + \beta_{k+1} p_k$$

$$k \leftarrow k + 1$$

end

4 Implementierung

4.1 Repräsentation des Berechnungsgebietes

Es handle sich bei $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ wieder um ein Berechnungsgebiet. Die Implementierung einer Finite-Elemente-Methode verlangt die Diskretisierung von $[\Omega]$, wie es in Abschnitt 3.1.2 beschrieben wurde. Das Ziel besteht darin, eine Klasse zu erstellen, die es ermöglicht, Berechnungsgebiete durch ein einfaches Dateiformat einzulesen, zu überprüfen und unter Umständen zu verfeinern. Eine weitere Klasse wird sich dann um die Konstruktion der Systemmatrizen und um die Berechnung des Zeitschrittes kümmern. Zunächst werden hierfür die grundlegendsten Strukturen eingeführt. Die simple Basisklasse **Domain_base** im folgenden Quelltext definiert Kanten und Dreiecke, die auf Eckpunkte referenzieren.

Code: **Domain_base**

```
namespace Fem {

    struct Domain_base {
        struct Edge;
        struct Primitive;

        Domain_base() = default;
        virtual ~Domain_base() = default;
        Domain_base(Domain_base&) = default;
        Domain_base& operator=(Domain_base&) = default;
        Domain_base(const Domain_base&) = default;
        Domain_base& operator=(const Domain_base&) = default;
    };

} // namespace Fem
```

Die Basisstruktur **Domain_base** wird im Namespace **Fem**, der auch für alle weiteren Codeausschnitte verwendet werden soll, konstruiert. In ihr werden die Strukturen **Edge** und **Primitive** für die Beschreibung der Kanten und Dreiecke deklariert. Die verschiedenen Konstruktoren und Zuweisungsoperatoren der Sprache C++ werden explizit durch den Compiler erzeugt. Der Destruktor wurde mit **virtual** gekennzeichnet, wie es für polymorphe Basisklassen angebracht ist [9, S. 40 ff]. In den folgenden beiden Codeausschnitten werden die Strukturen **Edge** und **Primitive** implementiert.

Code: **Edge**

```
namespace Fem {

    struct Domain_base::Edge : public std::array<int, 2> {
        struct Hash;
        struct Info;

        Edge(int v1, int v2);
```

4. IMPLEMENTIERUNG

```

};

Domain_base::Edge::Edge(int v1, int v2) {
    if (v1 > v2) std::swap(v1, v2);
    front() = v1;
    back() = v2;
}

struct Domain_base::Edge::Hash {
    std::size_t operator()(const Edge& edge) const {
        return edge[0] ^ (edge[1] << 1);
    }
};

struct Domain_base::Edge::Info {
    int insertions = 0;
    bool is_neumann_boundary = false;
};

} // namespace Fem

```

Code: **Primitive**

```

namespace Fem {

struct Domain_base::Primitive : public std::array<int, 3> {
    struct Hash;
    struct Info;

    Primitive(int v1, int v2, int v3);
};

Domain_base::Primitive::Primitive(int v1, int v2, int v3) {
    // insertion sort indices
    if (v1 > v2) std::swap(v1, v2);
    if (v2 > v3) std::swap(v2, v3);
    if (v1 > v2) std::swap(v1, v2);
    data()[0] = v1;
    data()[1] = v2;
    data()[2] = v3;
}

struct Domain_base::Primitive::Hash {
    std::size_t operator()(const Primitive& primitive) const {
        return primitive[0] ^ (primitive[1] << 1) ^ (primitive[2] << 2);
    }
};

struct Domain_base::Primitive::Info {
    int insertions = 0;
};

} // namespace Fem

```

Edge und **Primitive** sind analog zueinander aufgebaut. Bei beiden handelt es sich um eine Spezialisierung von `std::array<int, ...>` mit zwei oder drei Indizes. Diese Indizes werden später auf die Eckpunkte des Berechnungsgebietes verweisen. Die Konstruktoren stellen sicher, dass die Indizes sortiert übergeben werden. In beiden Fällen wird dabei eine explizite Formulierung des »Insertion Sort«-Verfahrens verwendet. Innerhalb der Datenstrukturen werden zwei weitere Datenstrukturen **Hash** und **Info** deklariert, die für die Verwendung mit der »Hash Map«

`std::unordered_map` vorgesehen sind. Eine »Hash Map« ist eine der effizientesten Varianten, um die Ränder eines Gebietes automatisch zu bestimmen. Zudem lässt sich durch diese sicher stellen, dass das konstruierte Gebiet fehlerfrei ist.

Die eigentliche Datenstruktur `Domain` eines Berechnungsgebietes ist dann eine Spezialisierung der Basisklasse `Domain_base` und nutzt demnach deren Kanten und Dreiecke. Der Datentyp der Eckpunkte wird durch ein »Template« variabel gehalten, sodass zum Beispiel der Austausch von Gleitkommazahlen mit einfacher Genauigkeit durch Gleitkommazahlen mit doppelter Genauigkeit kein Problem darstellt. Das Interface der Klasse ergibt sich aus Funktionen, um Daten zu lesen und fehlerfrei hinzuzufügen. Um dies auch in der Implementierung zu gewährleisten, speichert jede »Hash Map«, wie oft ein Dreieck oder eine Kante bereits hinzugefügt wurde. Auf der Basis dieser Zahl ist die Klasse dann in der Lage, zwischen Randkanten und inneren Kanten zu unterscheiden und fehlerhafte Berechnungsgebiete zu verhindern. Die folgenden beiden Codeausschnitte zeigen eine genaue Implementierung der wichtigsten Funktionen. In Abbildung 9 sind als Beispiel verschiedene Testgebiete durch die Klasse `Domain` eingelesen und verarbeitet worden.

Code: `Domain`

```
namespace Fem {

template <class T>
class Domain : public Domain_base {
public:
    using Vertex = T;
    using Edge = Domain_base::Edge;
    using Primitive = Domain_base::Primitive;
    using Quad = std::array<int, 4>

    Domain() = default;
    virtual ~Domain() = default;
    Domain(Domain&&) = default;
    Domain& operator=(Domain&&) = default;
    Domain(const Domain&) = default;
    Domain& operator=(const Domain&) = default;

    const auto& vertex_data() const { return vertex_data_; }
    const auto& primitive_data() const { return primitive_data_; }
    const auto& primitive_map() const { return primitive_map_; }
    const auto& edge_map() const { return edge_map_; }

    auto error_code(const Primitive& primitive) const;
    bool is_valid(const Primitive& primitive) const;
    void validate(const Primitive& primitive) const;

    Domain& add_vertex(const Vertex& vertex);
    Domain& operator<<(const Vertex& vertex) { return add_vertex(vertex); }
    Domain& add_primitive(const Primitive& primitive);
    Domain& operator<<(const Primitive& primitive) {
        return add_primitive(primitive);
    }
    Domain& add_quad(const Quad& quad);
    Domain& operator<<(const Quad& quad) { return add_quad(quad); }

    Domain& set_dirichlet_boundary(const Edge& edge);
    Domain& set_neumann_boundary(const Edge& edge);

    Domain& subdivide();
}
```

4. IMPLEMENTIERUNG

```

private:
    std::vector<Vertex> vertex_data_;
    std::vector<Primitive> primitive_data_;
    std::unordered_map<Edge, typename Edge::Info, typename Edge::Hash> edge_map_;
    std::unordered_map<Primitive, typename Primitive::Info,
                      typename Primitive::Hash>
        primitive_map_;
};

} // namespace Fem

```

Code: Domain Implementation

```

namespace Fem {

template <class Vertex>
Domain<Vertex>& Domain<Vertex>::add_vertex(const Vertex& vertex) {
    vertex_data_.push_back(vertex);
    return *this;
}

template <class Vertex>
Domain<Vertex>& Domain<Vertex>::add_primitive(const Primitive& primitive) {
    validate(primitive);

    ++edge_map_[Edge{primitive[0], primitive[1]}].insertions;
    ++edge_map_[Edge{primitive[0], primitive[2]}].insertions;
    ++edge_map_[Edge{primitive[1], primitive[2]}].insertions;

    ++primitive_map_[primitive].insertions;
    primitive_data_.push_back(primitive);

    return *this;
}

template <class Vertex>
Domain<Vertex>& Domain<Vertex>::add_quad(const Quad& quad) {
    Primitive primitive_1{quad[0], quad[1], quad[2]};
    Primitive primitive_2{quad[0], quad[2], quad[3]};

    validate(primitive_1);
    validate(primitive_2);

    add_primitive(primitive_1);
    add_primitive(primitive_2);

    return *this;
}

template <typename Vertex>
Domain<Vertex>& Domain<Vertex>::set_dirichlet_boundary(const Edge& edge) {
    auto& info = edge_map_.at(edge);
    if (info.insertions != 1)
        throw std::invalid_argument(
            "Could not set Dirichlet boundary! Given edge is an inner edge.");
    info.is_neumann_boundary = false;
    return *this;
}

template <typename Vertex>
Domain<Vertex>& Domain<Vertex>::set_neumann_boundary(const Edge& edge) {
    auto& info = edge_map_.at(edge);
    if (info.insertions != 1)
        throw std::invalid_argument(
            "Could not set Dirichlet boundary! Given edge is an inner edge.");
    info.is_neumann_boundary = true;
    return *this;
}

} // namespace Fem

```

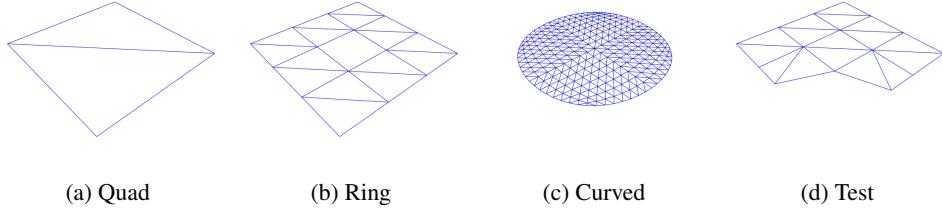


Abbildung 9: Die Abbildungen zeigen verschiedene Beispiele für Berechnungsgebiete, die mithilfe der Klasse **Domain** eingelesen und verarbeitet wurden.

Für die späteren Messungen ist es von großem Nutzen, die Triangulierung eines Gebietes zu verfeinern. Hierfür enthält die Klasse **Domain** eine Funktion **subdivide** die eine Subdivision des Berechnungsgebietes durchführt. Dabei ermittelt sie für jede vorhandene Kante einen neuen Eckpunkt und fügt dann auf der Basis der alten Dreiecke neue feinere Dreiecke hinzu. Die unterteilten Kanten behalten dabei ihre Randeigenschaften. In Abbildung 10 wird dieses Verfahren schematisch noch einmal an einem Dreieck demonstriert.

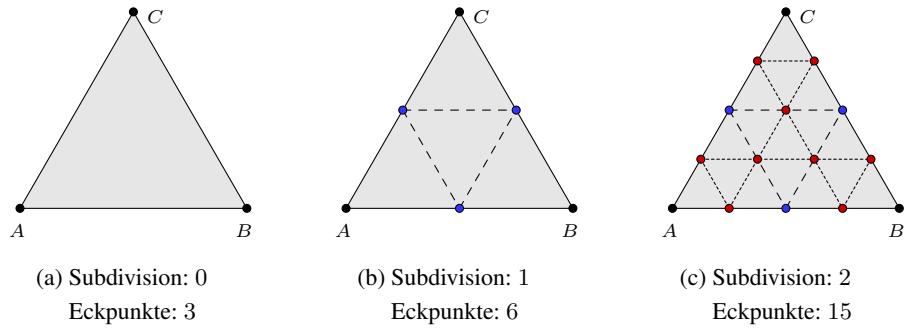


Abbildung 10: Die Skizzen zeigen unterschiedliche Subdivisions eines Dreiecks, definiert durch die Eckpunkte A , B und C . Zunächst wird jede Kante unterteilt. Danach können die Dreiecke durch feinere Dreiecke ersetzt werden.

Im folgenden Quelltext wird sich bereits auf das existierende Interface der Klasse **Domain** gestützt. Die Funktion **subdivide** bedingt, dass es sich bei dem gegebenen Template-Parameter um eine Klasse handeln muss, die einfache Vektorarithmetik unterstützt. Der Algorithmus stellt nicht sicher, dass Eckpunkte, die nah beieinander liegen, auch im Speicher nah beieinander liegen, wodurch dessen Formulierung simpler ausfällt. Die Anwendung von **subdivide** auf die Szenen aus Abbildung 9 ist in Abbildung 11 zu sehen.

4. IMPLEMENTIERUNG

Code: subdivide

```

namespace Fem {

template <class Vertex>
Domain<Vertex>& Domain<Vertex>::subdivide() {
    decltype(edge_map_) edge_subdivide_map;
    edge_subdivide_map.swap(edge_map_);

    decltype(primitive_data_) old_primitive_data;
    old_primitive_data.swap(primitive_data_);
    primitive_map_.clear();

    std::vector<typename decltype(edge_map_)::value_type> boundary_edges;

    for (auto& pair : edge_subdivide_map) {
        if (pair.second.insertions == 1) {
            boundary_edges.push_back(pair);
            boundary_edges.back().second.insertions = vertex_data_.size();
        }
    }

    pair.second.insertions = vertex_data_.size();
    add_vertex(0.5f *
               (vertex_data_[pair.first[0]] + vertex_data_[pair.first[1]]));
}

for (auto& primitive : old_primitive_data) {
    const int index_01 =
        edge_subdivide_map.at({primitive[0], primitive[1]}).insertions;
    const int index_12 =
        edge_subdivide_map.at({primitive[1], primitive[2]}).insertions;
    const int index_20 =
        edge_subdivide_map.at({primitive[2], primitive[0]}).insertions;

    add_primitive(Primitive(primitive[0], index_01, index_20));
    add_primitive(Primitive(index_01, index_12, index_20));
    add_primitive(Primitive(index_01, primitive[1], index_12));
    add_primitive(Primitive(index_12, primitive[2], index_20));
}

for (auto& pair : boundary_edges) {
    if (pair.second.is_neumann_boundary) {
        set_neumann_boundary(Edge{pair.first[0], pair.second.insertions});
        set_neumann_boundary(Edge{pair.first[1], pair.second.insertions});
    } else {
        set_dirichlet_boundary(Edge{pair.first[0], pair.second.insertions});
        set_dirichlet_boundary(Edge{pair.first[1], pair.second.insertions});
    }
}

return *this;
}

} // namespace Fem

```

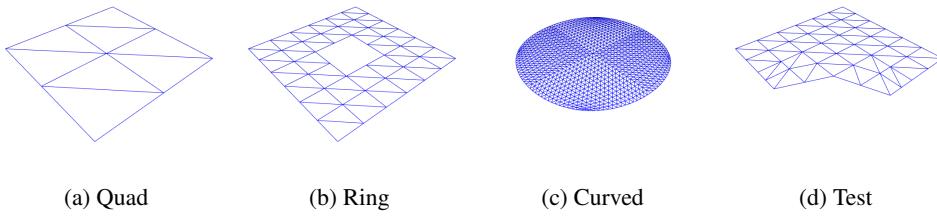


Abbildung 11: Die Abbildungen zeigen verschiedene Beispiele für Berechnungsgebiete, die mithilfe der Klasse `Domain` eingelesen und dann durch die Funktion `subdivide` verfeinert wurden.

Um ein Berechnungsgebiet außerhalb des Quelltextes leicht zu beschreiben, wurde ein einfaches ASCII-basiertes Dateiformat definiert. Im Folgenden wird es anhand eines Beispielmodells, welches [1] entnommen wurde, demonstriert. Hierbei steht **v** für die Definition eines Eckpunktes, **p** für die Definition eines Dreiecks, **q** für die Definition eines Vierecks, **d** für eine Dirichlet-Kante und **n** für eine Neumann-Kante. Beim Einlesen einer Datei wird durch das Interface der Klasse **Domain** sicher gestellt, dass es sich um korrekt konstruierte Gebiete handelt.

Zeile 1-13	Zeile 14-26	Zeile 27-39
<pre> v 0 0 v 1 0 v 1.59 0 v 2 1 v 3 1.41 v 3 2 v 3 3 v 2 3 v 1 3 v 0 3 v 0 2 v 0 1 v 1 1 </pre>	<pre> v 1 2 v 2 2 p 2 3 13 p 3 4 13 p 4 5 15 p 5 6 15 q 1 2 13 12 q 12 13 14 11 q 13 4 15 14 q 11 14 9 10 q 14 15 8 9 q 15 6 7 8 </pre>	<pre> n 5 6 n 6 7 n 1 2 n 2 3 d 3 4 d 4 5 d 7 8 d 8 9 d 9 10 d 10 11 d 11 12 d 12 1 </pre>

4.2 Aufbau des Systems

Nachdem nun eine robuste Struktur für das Einlesen, Bearbeiten und Speichern eines Berechnungsgebietes entwickelt wurde, ist ein System nötig, welches die Finite-Elemente-Methode anwendet und die Lösung der Wellengleichung für gegebene Rand- und Anfangsdaten simuliert.

Ein solches System soll zunächst auf der CPU implementiert werden, um die Anschaulichkeit zu gewährleisten. Hierfür wird die Klasse **Cpu_wave_system** eingeführt. Sie verwendet die Bibliothek »Eigen«, die ein abstraktes und leicht verständliches Interface für alle grundlegenden Matrix-Vektor-Operationen anbietet [19].

Code: **Cpu_wave_system**

```

namespace Fem {

template <typename Domain>
class Cpu_wave_system {
public:
    using value_type = typename Domain::Vertex::value_type;
    using Field = std::vector<value_type>;
    using Matrix = Eigen::SparseMatrix<value_type, Eigen::RowMajor>;
    using Permutation = std::vector<int>;

    Cpu_wave_system(const Domain& domain);
    ~Cpu_wave_system();

    const Permutation& permutation() const { return permutation_; }

    template <typename Iterator>
    Cpu_wave_system& copy_wave(Iterator wave_begin) const;

    template <typename Iterator>
    Cpu_wave_system& initial_state(Iterator wave_begin, Iterator evolution_begin);
}

```

4. IMPLEMENTIERUNG

```
Cpu_wave_system& advance(value_type dt);

private:
Field wave_;
Field evolution_;
Matrix mass_matrix_;
Matrix stiffness_matrix_;
Matrix boundary_mass_matrix_;
Matrix boundary_stiffness_matrix_;
Permutation permutation_;
};

} // namespace Fem
```

Das Interface der Klasse **Cpu_wave_system** besteht grundlegend aus der Funktion **initial_state**, die es ermöglicht, die Anfangs- und Randbedingungen zu setzen, und einer Funktion **advance**, die das gesamte System einen Zeitschritt voranbringt. Systemmatrizen der Finite-Elemente-Methode, sowie die Welle und deren Ableitungen, werden zwischengespeichert, um unnötige Berechnungsschritte zu verhindern. Des Weiteren sollen bei der Konstruktion des Systems Eckpunkte, die auf dem Rand des Gebietes liegen, von den inneren Eckpunkten separiert werden. Hierfür wird das Array der Eckpunkte innerhalb des Systems permutiert. Die Permutation wird für den Zugriff von außen gespeichert. Im weiteren Verlauf sollen die Neumann-Randbedingungen auf Null gesetzt werden, sodass die Konstruktion der Neumann-Rand-Matrix nicht notwendig ist. Diese ließe sich aber analog zu den anderen Systemmatrizen durch eine Iteration über alle Neumann-Kanten konstruieren.

Auf der GPU ist die Verwendung von »Eigen« nicht möglich. Aus diesem Grund ist eine explizite Implementierung des CSR-Formates auf dem DRAM der GPU nötig. Das Interface der Klasse **Gpu_wave_system** unterscheidet sich im Wesentlichen nicht von dem der Klasse **Cpu_wave_system**. Um den Zugriff auf die Daten im DRAM der GPU zu ermöglichen, werden Zeiger deklariert, die später einem Kernel der GPU übergeben werden. Möchte man die Wellenfunktion auslesen, so müssen deren Daten vom DRAM wieder in den RAM übertragen werden. Eben deshalb wird eine Funktion **copy_wave** deklariert.

Code: GPU Wave system

```
namespace Fem {

template <typename Domain>
class Gpu_wave_system {
public:
    using value_type = typename Domain::Vertex::value_type;
    using Permutation = std::vector<int>

    Gpu_wave_system(const Domain& domain);
    ~Gpu_wave_system();

    const Permutation& permutation() const { return permutation_; }
    template <typename Iterator>
    Gpu_wave_system& copy_wave(Iterator wave_begin) const;
    template <typename Iterator>
```

```

Gpu_wave_system& initial_state(Iterator wave_begin, Iterator evolution_begin);
Gpu_wave_system& advance(value_type dt);

private:
Permutation permutation_;

// CSR format of inner matrices
value_type* mass_values_;
value_type* stiffness_values_;
int* row_cdf_;
int* col_index_;
int nnz_;

// CSR format of boundary matrices
value_type* boundary_mass_values_;
value_type* boundary_stiffness_values_;
int* boundary_row_cdf_;
int* boundary_col_index_;
int boundary_nnz_;

// handlers for wave data
value_type* wave_;
value_type* evolution_;

// inner matrix dimension: inner_dimension x inner_dimension
// boundary matrix dimension: inner_dimension x boundary_dimension
int inner_dimension_;
int boundary_dimension_;

// values to efficiently launch GPU kernel
int threads_per_block_;
int blocks_;

// preallocated data for faster computation
// of conjugate gradient method
value_type* tmp_p_;
value_type* tmp_r_;
value_type* tmp_y_;
};

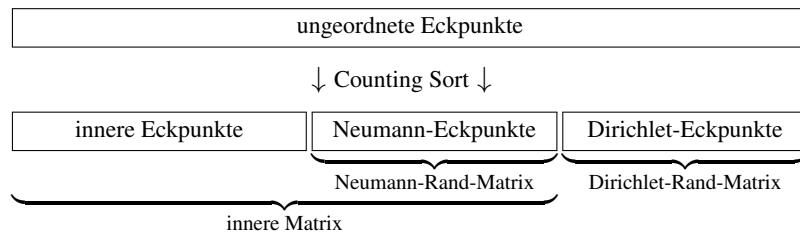
} // namespace Fem

```

4.3 Konstruktion der Systemmatrizen

Die Konstruktion der Systemmatrizen muss in dieser Arbeit für jedes Berechnungsgebiet nur ein einziges Mal ausgeführt werden. Aus diesem Grund soll hier auf die Implementierung einer parallelisierten Methode auf der GPU verzichtet werden.

Für eine effiziente Berechnung des Zeitschrittes sollen vor der eigentlichen Konstruktion alle Eckpunkte durch »Counting Sort« sortiert werden. Das folgende Schema veranschaulicht diese Überlegung und zeigt zudem die Bereiche der Indizes an, auf die die unterschiedlichen Systemmatrizen zugreifen. Die Sortierung der Eckpunkte steigert die Performance des Matrix-Vektor-Produktes, welches nun bei der Berechnung linear auf dem Speicher arbeitet, und vereinfacht zudem das Interface.



Für die Simulation der Wellengleichung mit natürlichem Neumann-Rand werden vier Systemmatrizen benötigt. Diese teilen sich auf in die inneren Matrizen, die die Wechselwirkung freier Eckpunkte untereinander beschreiben, und Randmatrizen, die die Wechselwirkung des Dirichlet-Randes mit inneren Eckpunkten beschreiben. Nach der Berechnung der Permutation durch »Counting Sort«, wird über alle Dreiecke iteriert. Jedes Dreieck liefert die Matrix eines finiten Elementes, wie in Abschnitt 3.1.4 beschrieben wurde. Diese Matrizen werden entsprechend dem Standardverfahren von »Eigen« akkumuliert und zu den Systemmatrizen zusammengesetzt [19].

Code: **Cpu_wave_system** Constructor

```

namespace Fem {

template <typename Domain>
Cpu_wave_system<Domain>::Cpu_wave_system(const Domain& domain) {
  using Vertex = typename Domain::Vertex;

  // mark Dirichlet vertices
  std::vector<int> is_boundary(domain.vertex_data().size(), 0);
  for (const auto& pair : domain.edge_map()) {
    if (pair.second.insertions != 1 || pair.second.is_neumann_boundary)
      continue;
    is_boundary[pair.first[0]] = 1;
    is_boundary[pair.first[1]] = 1;
  }

  // construct the permutation to separate Dirichlet vertices
  // count inner and Dirichlet vertices
  int vertex_count[2] = {0, 0};
  for (auto i = 0; i < is_boundary.size(); ++i) ++vertex_count[is_boundary[i]];
  // move indices to right position
  int inner_vertex_count = vertex_count[0];
  int boundary_vertex_count = vertex_count[1];
  vertex_count[0] = 0;
  vertex_count[1] = inner_vertex_count;
  permutation.resize(is_boundary.size());
  for (auto i = 0; i < is_boundary.size(); ++i) {
    permutation[vertex_count[is_boundary[i]]] = i;
    ++vertex_count[is_boundary[i]];
  }
  // construct inverse permutation
  std::vector<int> inverse_permutation(permutation.size());
  for (auto i = 0; i < permutation.size(); ++i) {
    inverse_permutation[permutation[i]] = i;
  }

  // construct triplets for inner and boundary matrices
  std::vector<Eigen::Triplet<value_type>> stiffness_triplets;
  std::vector<Eigen::Triplet<value_type>> boundary_stiffness_triplets;
  std::vector<Eigen::Triplet<value_type>> mass_triplets;
  std::vector<Eigen::Triplet<value_type>> boundary_mass_triplets;

  for (const auto& primitive : domain.primitive_data()) {
    Vertex edge[3];
    for (auto i = 0; i < 3; ++i) {
      edge[i] = domain.vertex_data()[primitive[(i + 1) % 3]] -
        domain.vertex_data()[primitive[i]];
    }

    const float dot_product = edge[0].dot(edge[2]);
    const float area =
      0.5 * std::sqrt(edge[0].squaredNorm() * edge[2].squaredNorm() -
                      dot_product * dot_product);
    const float inverse_area_4 = 0.25 / area;

    // diagonal entries
    for (auto i = 0; i < 3; ++i) {
      stiffness_triplets.push_back(Eigen::Triplet<value_type>(i, i,
        inverse_area_4));
      if (is_boundary[i])
        boundary_stiffness_triplets.push_back(Eigen::Triplet<value_type>(i, i,
          inverse_area_4));
    }

    // off-diagonal entries
    for (auto i = 0; i < 3; ++i) {
      for (auto j = i + 1; j < 3; ++j) {
        stiffness_triplets.push_back(Eigen::Triplet<value_type>(i, j,
          -inverse_area_4));
        boundary_stiffness_triplets.push_back(Eigen::Triplet<value_type>(i, j,
          -inverse_area_4));
      }
    }
  }
}

```

```

    if (is_boundary[primitive[i]]) continue;
    const value_type stiffness_value =
        inverse_area_4 * edge[(i + 1) % 3].squaredNorm();
    const value_type mass_value = area / 6.0;
    const int index = inverse_permutation[primitive[i]];
    stiffness_triplets.push_back({index, index, stiffness_value});
    mass_triplets.push_back({index, index, mass_value});
}

// lower triangle
for (unsigned int i = 0; i < 3; ++i) {
    if (is_boundary[primitive[i]]) continue;
    const int index_i = inverse_permutation[primitive[i]];

    for (unsigned int j = 0; j < i; ++j) {
        const value_type stiffness_value =
            inverse_area_4 * edge[(i + 1) % 3].dot(edge[(j + 1) % 3]);
        const value_type mass_value = area / 12.0;

        if (is_boundary[primitive[j]]) {
            // boundary matrices are not symmetric
            const int index_j =
                inverse_permutation[primitive[j]] - inner_vertex_count;
            boundary_stiffness_triplets.push_back(
                {index_i, index_j, stiffness_value});
            boundary_mass_triplets.push_back({index_i, index_j, mass_value});
        } else {
            // inner matrices have to be symmetric
            const int index_j = inverse_permutation[primitive[j]];
            stiffness_triplets.push_back({index_i, index_j, stiffness_value});
            stiffness_triplets.push_back({index_j, index_i, stiffness_value});
            mass_triplets.push_back({index_i, index_j, mass_value});
            mass_triplets.push_back({index_j, index_i, mass_value});
        }
    }
}

// construct matrices from triplets
stiffness_matrix = Eigen::SparseMatrix<value_type, Eigen::RowMajor>(
    inner_vertex_count, inner_vertex_count);
stiffness_matrix.setFromTriplets(stiffness_triplets.begin(),
                                stiffness_triplets.end());

boundary_stiffness_matrix = Eigen::SparseMatrix<value_type, Eigen::RowMajor>(
    inner_vertex_count, boundary_vertex_count);
boundary_stiffness_matrix.setFromTriplets(boundary_stiffness_triplets.begin(),
                                         boundary_stiffness_triplets.end());

mass_matrix = Eigen::SparseMatrix<value_type, Eigen::RowMajor>(
    inner_vertex_count, inner_vertex_count);
mass_matrix.setFromTriplets(mass_triplets.begin(), mass_triplets.end());

boundary_mass_matrix = Eigen::SparseMatrix<value_type, Eigen::RowMajor>(
    inner_vertex_count, boundary_vertex_count);
boundary_mass_matrix.setFromTriplets(boundary_mass_triplets.begin(),
                                     boundary_mass_triplets.end());
}
} // namespace Fem

```

Für die GPU müssen die auf der CPU konstruierten Daten der dünnbesetzten Matrizen noch auf den DRAM übertragen werden. Nützlich dabei ist, dass die beiden inneren Matrizen das gleiche Muster aufweisen. Ebenso gilt dies für die beiden Randmatrizen. Die vier CSR-Matrizen lassen sich also zu zwei kompletten CSR-Matrizen und zwei zusätzlichen Wertarrays komprimieren. Möchte man die Neumann-Rand-Matrix implementieren, so muss eine weitere komplett CSR-Matrix eingeführt werden, da diese ein anderes Muster als die zuvor genannten Matrizen aufweist.

4. IMPLEMENTIERUNG

Während der Konstruktion wird zudem die maximale Anzahl von Threads auf einem Block der verwendeten GPU ausgelesen. Mithilfe dieses Wertes wird eine optimale Anzahl von Threads auf den verschiedenen Blocks der GPU generiert. Die Aufteilung von Threads auf verschiedene Blöcke wird in [6, 7, 15] diskutiert. In [2] werden für verschiedene Problemstellungen zudem diverse Vorteile durch die Verwendung der »Shared Memory« innerhalb eines Blocks für diese Methode erläutert.

Code: `Gpu_wave_system` Constructor

```

namespace Fem {

template <typename Domain>
Gpu_wave_system<Domain>::Gpu_wave_system(const Domain& domain) {
    // construct system matrix and permutation
    // as done in Cpu_wave_system
    // ...

    // allocate memory on DRAM of GPU
    cudaMalloc((void**)&mass_values_, nnz_ * sizeof(value_type));
    cudaMalloc((void**)&stiffness_values_, nnz_ * sizeof(value_type));
    cudaMalloc((void**)&row_cdf_, (inner_dimension_ + 1) * sizeof(int));
    cudaMalloc((void**)&col_index_, nnz_ * sizeof(int));

    cudaMalloc((void**)&boundary_mass_values_,
               boundary_nnz_ * sizeof(value_type));
    cudaMalloc((void**)&boundary_stiffness_values_,
               boundary_nnz_ * sizeof(value_type));
    cudaMalloc((void**)&boundary_row_cdf_, (inner_dimension_ + 1) * sizeof(int));
    cudaMalloc((void**)&boundary_col_index_, boundary_nnz_ * sizeof(int));

    cudaMalloc((void**)&wave_,
               (inner_dimension_ + boundary_dimension_) * sizeof(value_type));
    cudaMalloc((void**)&evolution_,
               (inner_dimension_ + boundary_dimension_) * sizeof(value_type));

    cudaMalloc((void**)&tmp_p_, inner_dimension_ * sizeof(value_type));
    cudaMalloc((void**)&tmp_r_, inner_dimension_ * sizeof(value_type));
    cudaMalloc((void**)&tmp_y_, inner_dimension_ * sizeof(value_type));

    // transfer data from RAM to DRAM
    cudaMemcpy(mass_values_, mass_matrix.valuePtr(), nnz_ * sizeof(value_type),
               cudaMemcpyHostToDevice);
    cudaMemcpy(stiffness_values_, stiffness_matrix.valuePtr(),
               nnz_ * sizeof(value_type), cudaMemcpyHostToDevice);
    cudaMemcpy(row_cdf_, mass_matrix.outerIndexPtr(),
               (inner_dimension_ + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(col_index_, mass_matrix.innerIndexPtr(), nnz_ * sizeof(int),
               cudaMemcpyHostToDevice);

    cudaMemcpy(boundary_mass_values_, boundary_mass_matrix.valuePtr(),
               boundary_nnz_ * sizeof(value_type), cudaMemcpyHostToDevice);
    cudaMemcpy(boundary_stiffness_values_, boundary_stiffness_matrix.valuePtr(),
               boundary_nnz_ * sizeof(value_type), cudaMemcpyHostToDevice);
    cudaMemcpy(boundary_row_cdf_, boundary_mass_matrix.outerIndexPtr(),
               (inner_dimension_ + 1) * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(boundary_col_index_, boundary_mass_matrix.innerIndexPtr(),
               boundary_nnz_ * sizeof(int), cudaMemcpyHostToDevice);

    // get properties of GPU to compute blocks and threads_per_block
    cudaDeviceProp property;
    cudaGetDeviceProperties(&property, 0);
    threads_per_block_ = property.maxThreadsPerBlock;
    blocks_ = (inner_dimension + threads_per_block - 1) / threads_per_block;
}

} // namespace Fem

```

Da fast die gesamten Daten auf dem DRAM der GPU über herkömmliche Zeiger verwaltet werden, sollte der Destruktor der Klasse `Gpu_wave_system` nach dem RAII-Prinzip den allozierten Speicher wieder freigeben [17]. Der folgende Codeausschnitt zeigt dies. Die Funktionen, um die Anfangsdaten zu setzen und die Wellendaten auszulesen, müssen ebenfalls die Daten zwischen dem RAM und dem DRAM transportieren.

Code: `Gpu_wave_system` Destructor

```

namespace Fem {
    template <typename Domain>
    Gpu_wave_system<Domain>::~Gpu_wave_system() {
        cudaFree(mass_values_);
        cudaFree(stiffness_values_);
        cudaFree(row_cdf_);
        cudaFree(col_index_);

        cudaFree(boundary_mass_values_);
        cudaFree(boundary_stiffness_values_);
        cudaFree(boundary_row_cdf_);
        cudaFree(boundary_col_index_);

        cudaFree(wave_);
        cudaFree(evolution_);

        cudaFree(tmp_p_);
        cudaFree(tmp_x_);
        cudaFree(tmp_y_);
    }
}
// namespace Fem

```

4.4 Berechnung eines Zeitschrittes

Die Berechnung eines Zeitschrittes erfordert die Konstruktion der rechten Seite des linearen Gleichungssystems, die sich nach Abschnitt 3.1.4 aus vier Matrix-Vektor-Produkten zusammensetzt. Die Bibliothek »Eigen« ist in der Lage, diese Berechnungen durch ein abstraktes Vorgehen in anschaulicher Weise darzustellen. Mithilfe des in »Eigen« implementierten CG-Verfahrens wird dann die Ableitung der Welle zum neuen Zeitpunkt berechnet. Anschließend kann mit dieser Ableitung die neue Welle generiert werden. Der folgende Codeausschnitt demonstriert dieses Vorgehen. Zu beachten ist, dass die Daten der Vektoren `wave_` und `evolution_` in der sortierten Form vorliegen und so eine einfache Verarbeitung durch `Eigen::Map` ermöglichen. Die Implementierung auf der CPU ist jedoch nicht vollständig optimiert, da das eigentliche Ziel darin besteht das Verfahren auf der GPU zu implementieren.

4. IMPLEMENTIERUNG

Code: `Cpu_wave_system::advance`

```
namespace Fem {

template <typename Domain>
Cpu_wave_system<Domain> & Cpu_wave_system<Domain> :: advance(value_type dt) {
    using Vector = Eigen::Matrix<value_type, Eigen::Dynamic, 1>;
    using Vector_map = Eigen::Map<Vector>;

    Vector_map x(wave_.data(), mass_matrix_.cols());
    Vector_map y(evolution_.data(), mass_matrix_.cols());
    Vector_map boundary_x(wave_.data(), boundary_mass_matrix_.cols());
    Vector_map boundary_y(evolution_.data(), boundary_mass_matrix_.cols());

    Vector rhs =
        mass_matrix_ * y + boundary_mass_matrix_ * boundary_y -
        dt * (stiffness_matrix_ * x + boundary_stiffness_matrix_ * boundary_x);
    Eigen::ConjugateGradient<Matrix> solver;
    solver.compute(mass_matrix_);
    y = solver.solve(rhs);
    x = x + dt * y;

    return *this;
}

} // namespace Fem
```

Um die Ausführung der Funktion `advance` auf der GPU zu verstehen, soll zunächst das CG-Verfahren aus Abschnitt 3.2.2 auf der CPU implementiert werden. Der Quelltext der Funktion `Gpu_wave_system::advance` beschreibt genau das gleiche Vorgehen, ist aber durch die Anwendung zweier GPU-Kernel und weiteren Funktionen der Bibliothek »Thrust« komplizierter notiert [6]. Der folgende Codeausschnitt beschreibt das CG-Verfahren unter Verwendung dreier temporärer Vektoren, um die Berechnung zu beschleunigen. Die Template-Parameter beschreiben hier Konzepte einer Matrix und eines Vektors, sodass dieser Algorithmus auf verschiedene Datentypen angewandt werden könnte.

Code: `conjugate_gradient`

```
template <typename Matrix, typename Vector>
void conjugate_gradient(const Matrix& A, Vector& x, const Vector& b) {
    constexpr float max_error = 1e-6f;

    Vector r = A * x - b;
    Vector p = -r;
    Vector y;
    float res = r.squaredNorm();

    for (auto i = 0; i < b.size(); ++i) {
        y = A * p;
        const float alpha = res / p.dot(y);
        x += alpha * p;
        r += alpha * y;
        const float new_res = r.squaredNorm();
        if (new_res <= max_error) break;
        const float beta = new_res / res;
        res = new_res;
        p = beta * p - r;
    }
}
```

Auf der GPU wird nun mithilfe von CUDA und »Thrust« genau wie auf der CPU zuerst die rechte Seite des linearen Gleichungssystems konstruiert und dann das CG-Verfahren angewendet. Im folgenden Quelltext werden für diese Aufgaben zudem zwei verschiedene Kernel `spmv_csr_kernel` und `rhs_kernel` verwendet. Der erste Kernel berechnet das Matrix-Vektor-Produkt einer im CSR-Format gespeicherten Matrix. Der zweite Kernel berechnet mit einem einzigen Aufruf die rechte Seite des linearen Gleichungssystems, um temporäre Speicherbewegungen.

Code: `Gpu_wave_system::advance`

```

namespace Fem {

template <typename Domain>
Gpu_wave_system<Domain>& Gpu_wave_system<Domain>::advance(value_type dt) {
    thrust::device_ptr<value_type> dev_evolution(evolution_);
    thrust::device_ptr<value_type> dev_wave(wave_);
    thrust::device_ptr<value_type> dev_tmp_y(tmp_y_);
    thrust::device_ptr<value_type> dev_tmp_p(tmp_p_);
    thrust::device_ptr<value_type> dev_tmp_r(tmp_r_);

    rhs_kernel<value_type><<<blocks_, threads_per_block>>>(
        inner_dimension_, mass_values_, stiffness_values_, row_cdf_, col_index_,
        boundary_mass_values_, boundary_stiffness_values_, boundary_row_cdf_,
        boundary_col_index_, wave_, evolution_, dt, tmp_r_);

    spmv_csr_kernel<value_type><<<blocks_, threads_per_block>>>(
        inner_dimension_, 1.0f, mass_values_, row_cdf_, col_index_, evolution_,
        -1.0f, tmp_r_);

    thrust::transform(dev_tmp_r, dev_tmp_r + inner_dimension_, dev_tmp_p,
                    thrust::negate<value_type>());

    value_type res = thrust::inner_product(
        dev_tmp_r, dev_tmp_r + inner_dimension_, dev_tmp_r, 0.0f);
    int it = 0;

    while ((it < 1 || res > 1e-6f) && it < inner_dimension_) {
        spmv_csr_kernel<value_type><<<blocks_, threads_per_block>>>(
            inner_dimension_, 1.0f, mass_values_, row_cdf_, col_index_, tmp_p_,
            0.0f, tmp_y_);

        const value_type tmp = thrust::inner_product(
            dev_tmp_p, dev_tmp_p + inner_dimension_, dev_tmp_y, 0.0f);
        const value_type alpha = res / tmp;

        thrust::transform(dev_tmp_p, dev_tmp_p + inner_dimension_, dev_evolution,
                        dev_evolution, axpy_functor<value_type>(alpha));
        thrust::transform(dev_tmp_y, dev_tmp_y + inner_dimension_, dev_tmp_r,
                        dev_tmp_r, axpy_functor<value_type>(alpha));

        const value_type new_res = thrust::inner_product(
            dev_tmp_r, dev_tmp_r + inner_dimension_, dev_tmp_r, 0.0f);
        const value_type beta = new_res / res;
        res = new_res;

        thrust::transform(dev_tmp_p, dev_tmp_p + inner_dimension_, dev_tmp_r,
                        dev_tmp_p, axpy_functor<value_type>(beta, -1.0f));
        ++it;
    }

    thrust::transform(dev_evolution, dev_evolution + inner_dimension_, dev_wave,
                    dev_wave, axpy_functor<value_type>(dt));
}

return *this;
}
} // namespace Fem

```

Die GPU-Kernel parallelisieren die Matrix-Vektor-Produkte, indem sie für jede Zeile der Matrix einen Thread verwenden, der ein Skalarprodukt bildet. Für das hier betrachtete zweidimensionale Wellenproblem ist dieser skalare Kernel ausreichend und effizient [2, 3]. Systeme zur Lösung komplexerer Probleme können von der Verwendung eines Vektor-Kernels, wie in [2] beschrieben, profitieren. Des Weiteren wurden zwei Funktoren **axpy_functor** und **axpby_functor** verwendet, die die typischen BLAS Routinen darstellen und aus der »Thrust«-Dokumentation entnommen wurden [6].

Code: GPU Kernel

```

namespace Fem {
namespace {

template <typename T>
struct axpy_functor : public thrust::binary_function<T, T, T> {
    using value_type = T;

    const value_type a;

    axpy_functor(value_type _a) : a(_a) {}

    __host__ __device__ value_type operator()(const value_type& x,
                                              const value_type& y) const {
        return a * x + y;
    }
};

template <typename T>
struct axpby_functor : public thrust::binary_function<T, T, T> {
    using value_type = T;

    const value_type a;
    const value_type b;

    axpby_functor(value_type _a, value_type _b) : a(_a), b(_b) {}

    __host__ __device__ value_type operator()(const value_type& x,
                                              const value_type& y) const {
        return a * x + b * y;
    }
};

template <typename value_type>
__global__ void rhs_kernel(
    int n, const value_type* mass_values, const value_type* stiffness_values,
    const int* row_cdf, const int* col_index,
    const value_type* boundary_mass_values,
    const value_type* boundary_stiffness_values, const int* boundary_row_cdf,
    const int* boundary_col_index, const value_type* wave_values,
    const value_type* evolution_values, value_type dt, value_type* output) {
    const int row = blockDim.x * blockIdx.x + threadIdx.x;

    if (row < n) {
        value_type mass_dot = 0;
        value_type stiffness_dot = 0;

        const int start = row_cdf[row];
        const int end = row_cdf[row + 1];
        for (int j = start; j < end; ++j) {
            mass_dot += mass_values[j] * evolution_values[col_index[j]];
            stiffness_dot += stiffness_values[j] * wave_values[col_index[j]];
        }

        const int boundary_start = boundary_row_cdf[row];
        const int boundary_end = boundary_row_cdf[row + 1];
        for (int j = boundary_start; j < boundary_end; ++j) {
    }
}
}
}

```

```

        mass_dot +=  

            boundary_mass_values[j] * evolution_values[n + boundary_col_index[j]];  

        stiffness_dot +=  

            boundary_stiffness_values[j] * wave_values[n + boundary_col_index[j]];  

    }  

    output[row] = mass_dot - dt * stiffness_dot;  

}  

}  

template <typename value_type>  

__global__ void spmv_csr_kernel(int n, value_type alpha,  

                                const value_type* values, const int* row_cdf,  

                                const int* col_index, const value_type* input,  

                                value_type beta, value_type* output) {  

const int row = blockDim.x * blockIdx.x + threadIdx.x;  

if (row < n) {  

    value_type dot = 0;  

    const int start = row_cdf[row];  

    const int end = row_cdf[row + 1];  

    for (int j = start; j < end; ++j) dot += values[j] * input[col_index[j]];  

    output[row] = beta * output[row] + alpha * dot;  

}
}  

} // namespace  

} // namespace Fem

```

4.5 Visualisierung

Für die Simulation der Wellengleichung reicht es an sich aus, das CG-Verfahren zu verwenden und das lineare Gleichungssystem zu lösen. Bei der berechneten Lösung handelt es sich um einen Vektor, der die Werte der Wellenfunktion über den einzelnen Eckpunkten speichert. Durch automatisierte Testverfahren ist der Computer teilweise in der Lage, die Ergebnisse auf Fehler zu überprüfen. Allerdings verändert sich eine Welle auf einem berandeten Gebiet mit fortschreitender Zeit auf immer komplexere Art und Weise. Aus diesem Grund stellt das subjektive Empfinden des Programmierers beziehungsweise Modellierers, welches bei der Betrachtung der Welle entsteht, ein weiteres Testverfahren dar, um die Korrektheit des Algorithmus unter gewissen Einschränkungen zu gewährleisten. Die Eigenschaften einer visualisierten Wellenfunktion sind für einen Menschen leichter nachzuvollziehen als eine reine Ansammlung von Daten. Abbildung 12 zeigt zum Beispiel numerische Fehler einer Wellensimulation, die durch die spezielle Form des Gitters auftreten und schwer durch automatisierte Tests zu finden sind.

Die Verwendung von OpenGL ermöglicht die dreidimensionale Darstellung von Objekten, die durch Dreiecke im dreidimensionalen Raum diskretisiert wurden. Die finiten Elemente des Berechnungsgebietes sind grundsätzlich im zweidimensionalen Raum definiert. Deshalb soll eine dritte Koordinate den Wert der Wellenfunktion an der entsprechenden Stelle eines jeden Eckpunktes beschreiben. Dies führt dazu, dass sich das angezeigte Berechnungsgebiet entsprechend der Welle verformt, sodass dessen

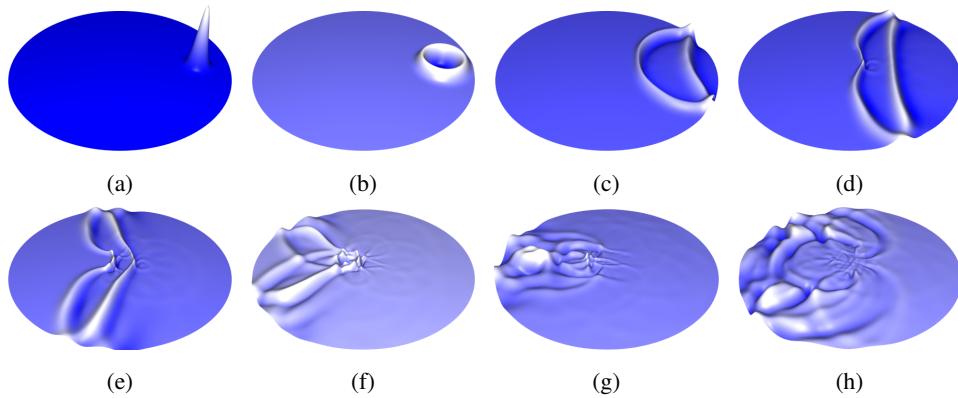


Abbildung 12: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf dem Gebiet »Circle«. Bei diesem Berechnungsgebiet handelt es sich um einen Kreis mit Neumann-Randbedingungen. Die spezielle Art der Triangulierung verursacht hier in der Nähe des Mittelpunktes numerische Fehler. Bereits im vierten Bild entsteht im Zentrum der Welle ein Knick, der das Ergebnis verfälscht.

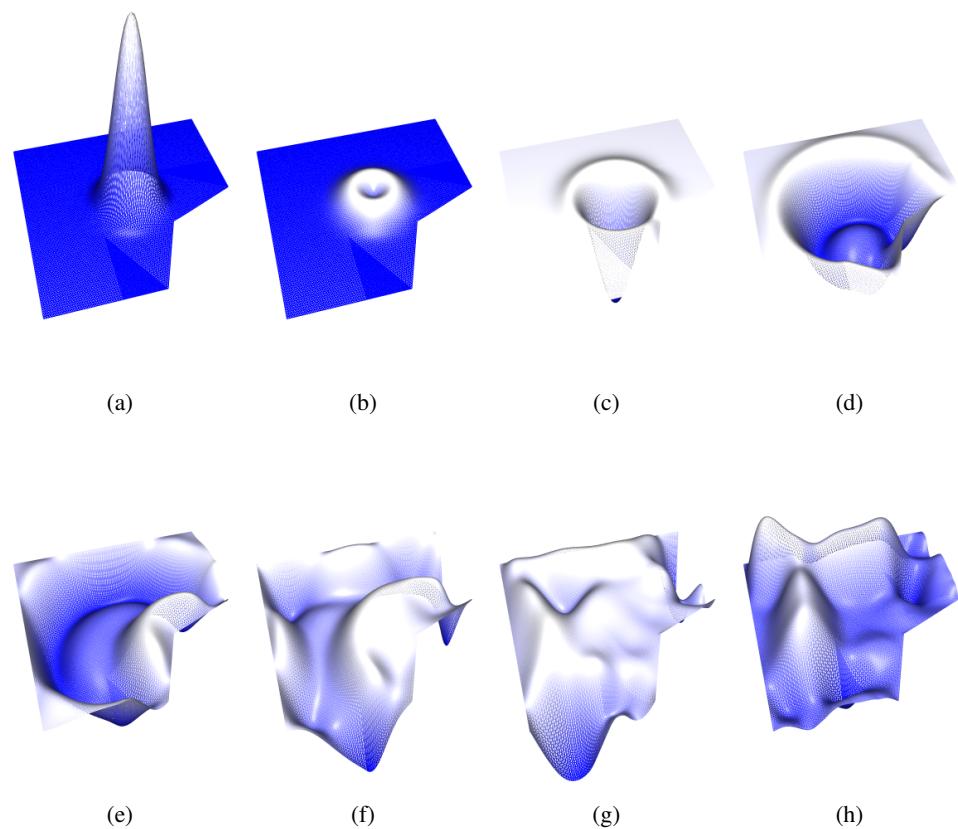


Abbildung 13: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf dem Gebiet »Test«. Dieses Gebiet wurde in [1] beschrieben. Es wurden dabei gemischte Randbedingungen gewählt.

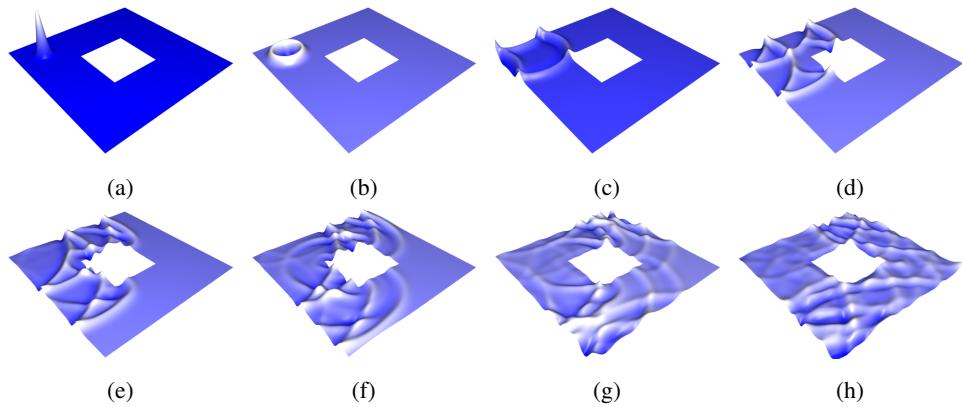


Abbildung 14: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf dem Gebiet »Ring«. Es handelt sich dabei um einen quadratischen Ring mit Neumann-Randbedingungen. Es lassen sich hier typische Effekte, wie Beugung und Interferenz, beobachten.

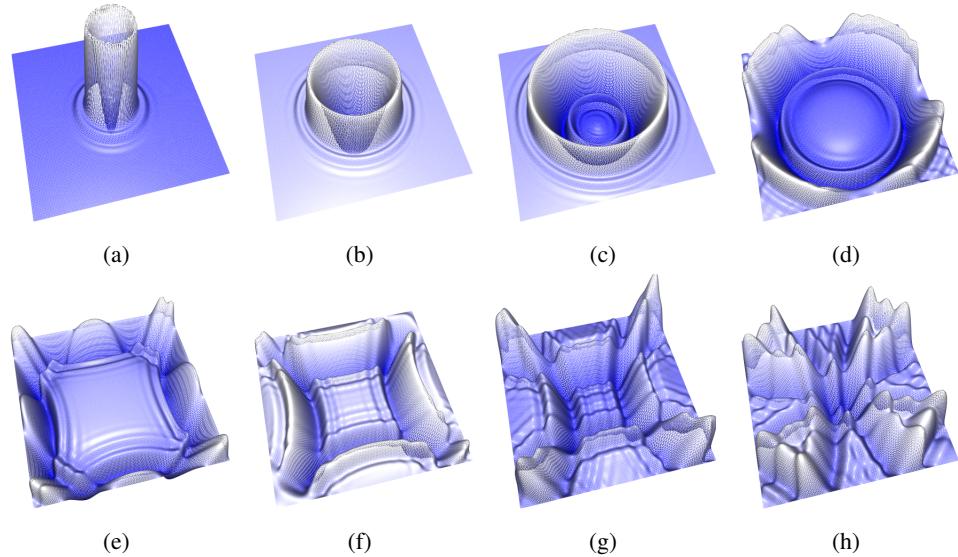


Abbildung 15: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf dem Gebiet »Quad«. Es handelt sich um ein quadratisches Berechnungsgebiet mit Dirichlet-Randbedingungen.

4. IMPLEMENTIERUNG

Aussehen mit der einer Wasseroberfläche verglichen werden kann. Um den Eindruck zu verstärken, wird für jeden Eckpunkt auf der Basis der benachbarten Dreiecke eine Normale berechnet, die das Gebiet wie eine differenzierbare Fläche erscheinen lässt. Weiterhin wird den einzelnen Werten der Welle noch eine Farbe zugewiesen. Damit fällt es dem Benutzer einfacher auch kleine Unterschiede wahrzunehmen. Die Abbildungen 13, 15, 14 und 12 zeigen die Visualisierung der berechneten Welle über dem entsprechenden Gebiet anhand ausgewählter Beispiele.

5 Simulation und Ergebnisse

Wie in den vorherigen Abschnitten gezeigt wurde, ist es möglich, eine Finite-Elemente-Methode sowohl auf der CPU als auch auf der GPU zu implementieren. Der dafür erforderliche Aufwand ist annähernd vergleichbar und ist durch die Verwendung von externen Bibliotheken, wie zum Beispiel CUSP, optimierbar. Die Entscheidung, welche der beiden Prozessorarten besser für ein solches Problem geeignet ist, kann anhand diverser Parameter getroffen werden. Einer der aussagekräftigsten Parameter hierzu ist die Performance des Programms.

Tabelle 1: Die Tabelle stellt Messwerte der Szene »Circle« dar. s bezeichnet dabei, wie oft die Funktion **subdivision** angewendet wurde, t_C steht für die Konstruktionszeit der Systemmatrizen, t_A bezeichnet die Dauer der Berechnung eines Zeitschrittes. Die Zeitspanne t_A wurde für die CPU und die GPU notiert. Für jede Modellunterteilung bezeichnet v die Anzahl der Eckpunkte, p die Anzahl der Dreiecke und e die Anzahl der Kanten. Die Mass Matrix eines jeden Systems besitzt die Dimension n und enthält z Werte, die nicht Null sind. Alle Messwerte wurden auf einem Computersystem mit einem Intel Core i7-7700K mit 4.2 GHz als CPU und einer NVIDIA GeForce GTX 1070 mit 8 GB DDR5 Speicher und PCIe 3.0 als GPU aufgenommen.

s	Modellparameter	Matrixparameter	t_C [ms]	t_A [ms]
0	$v = 5\,185$	$n = 5\,185$	CPU	85 ± 5
	$p = 10\,240$	$z = 36\,033$	GPU	—
	$e = 15\,424$			2.35 ± 0.05
1	$v = 20\,609$	$n = 20\,609$	CPU	100 ± 5
	$p = 40\,960$	$z = 143\,745$	GPU	—
	$e = 61\,568$			2.35 ± 0.05
2	$v = 82\,177$	$n = 82\,177$	CPU	150 ± 10
	$p = 163\,840$	$z = 574\,209$	GPU	—
	$e = 246\,016$			2.20 ± 0.05
3	$v = 328\,193$	$n = 328\,193$	CPU	435 ± 5
	$p = 655\,360$	$z = 2\,295\,297$	GPU	—
	$e = 983\,552$			3.00 ± 0.50
4	$v = 1\,311\,745$	$n = 1\,311\,745$	CPU	2000 ± 50
	$p = 2\,621\,440$	$z = 9\,178\,113$	GPU	—
	$e = 3\,933\,184$			17.0 ± 0.5
5	$v = 5\,244\,929$	$n = 5\,244\,929$	CPU	9500 ± 200
	$p = 10\,485\,760$	$z = 36\,706\,305$	GPU	—
	$e = 15\,730\,688$			56 ± 1

In Tabelle 1 werden diesbezüglich unter anderem die Zeiten für die Berechnung eines Zeitschrittes der beiden Prozessorarten für verschiedene Szenengrößen verglichen. Dabei wurde festgestellt, dass die Implementierung auf der GPU für die betrachteten

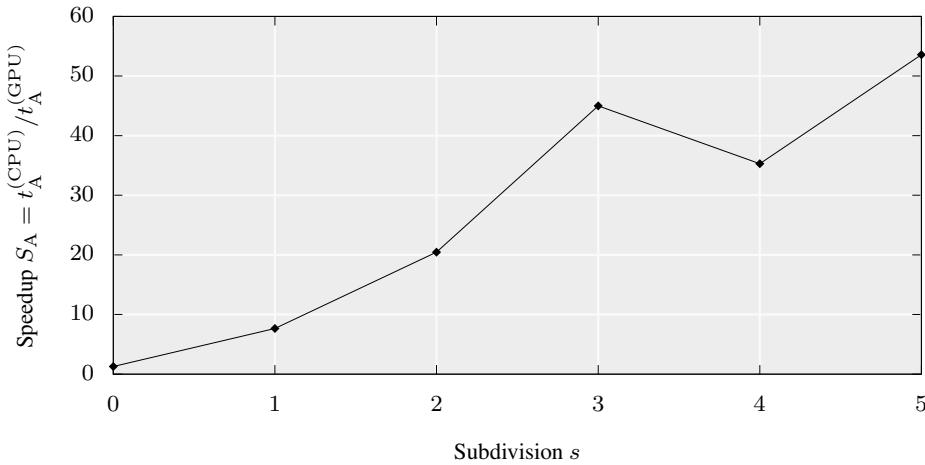


Abbildung 16: Die Abbildung zeigt den durch die GPU erreichten Speedup der Berechnung des Zeitschrittes für die verschiedenen Unterteilungen des gegebenen Modells aus Tabelle 1.

Modelle eine schnellere Berechnung ermöglicht im Vergleich zur CPU. Dies basiert im Wesentlichen auf der hohen Datenparallelität und der Möglichkeit der GPU, diese durch ihre extrem hohe Anzahl an Kernen auszunutzen. Zudem wurde die vollständige Auslastung der GPU erst für Szenen mit circa einer Million Eckpunkte und mehr erreicht. Dies bedeutet wiederum, dass die GPU für kleine Modelle nicht ausgelastet ist, und eine parallelisierte Implementierung auf CPU das Problem unter Umständen effizienter lösen könnte. Im Diagramm 16 ist der Effizienzgewinn (engl.: *speedup*) der GPU im Vergleich zur CPU noch einmal dargestellt.

Tabelle 1 zeigt weiterhin die Zeit, die benötigt wird, um die Systemmatrizen auf der CPU zu konstruieren. Der Zusammenhang zwischen Szenengröße und Konstruktionsdauer zeigt eine positive Korrelation. Je feiner die Auflösung des Berechnungsgebietes, desto größer wird die benötigte Konstruktionszeit. Diese simple Aussage ist der Tatsache geschuldet, dass die Konstruktion der Systemmatrizen auf der CPU nur seriell vollzogen wurde. Ein vergleichbarer Zusammenhang auf der GPU ist anhand der Messwerte nicht ersichtlich.

Ein Vergleich der Konstruktionsdauer mit der Dauer der Zeitschrittberechnung auf der CPU zeigt, dass die Generierung der Systemmatrizen für große Modelle circa dreimal länger benötigt als die Berechnung des Zeitschrittes. Kleinere Systeme weisen ein noch ungünstigeres Verhältnis auf, sodass die Konstruktion sogar bis zu 95% der Gesamtrechenzeit einnimmt. Dies stellt allerdings kein großes Problem dar, weil die Berechnung eines einzelnen Zeitschrittes beliebig oft wiederholt wird, während die Konstruktion nur einmalig ausgeführt wird.

Die ausgewählten numerischen Methoden, wie die Diskretisierung der Funktionenräume durch Hutfunktionen, das Speichern dünnbesetzter Matrizen im CSR-Format und das iterative Lösen des linearen Gleichungssystems durch das CG-Verfahren,

stellen nur eine Möglichkeit für die Implementierung einer Finite-Elemente-Methode dar. Dennoch zeigt sich, dass die Methoden für die Simulation der Wellengleichung nicht nur ausreichend, sondern auch effizient waren. Gerade bei den hier betrachteten zweidimensionalen Problemen stellen die Hutfunktionen die einfachste Form der Approximation dar und sind aus diesem Grund am schnellsten zu berechnen. Für dünnbesetzte Matrizen existiert zwar eine Vielzahl von Speicherformaten, jedoch ist das verwendete CSR-Format eine der kompaktesten und einfachsten Varianten. Die Effizienz der zugrundeliegenden Datenstruktur ist nur bis zu einem gewissen Grad von dem Muster der dünnbesetzten Matrix abhängig und liefert demzufolge eine konstante Performance für unterschiedliche Szenen. Auch für die Lösung von linearen Gleichungssystemen existiert eine Vielzahl von Algorithmen. Die hier verwendete Methode zeichnet sich durch ihren iterativen Charakter aus. Sie benötigt während der Berechnung nur einen moderaten Speicherplatz und löst auch sehr große Systeme im Zwei- und Dreidimensionalen effizient und unabhängig von der Permutation der Eckpunkte. Die Systemmatrizen sind positiv definit und symmetrisch. Das Verfahren nutzt diese Eigenschaften aus, um in wenigen Schritten zu einer Lösung zu gelangen. Des Weiteren lässt es sich durch sogenannte Vorkonditionierer verbessern.

Jede hier beschriebene Methode wurde anhand der Wellengleichung implementiert. Die Behandlung der Wellengleichung kann auf weitere partielle Differentialgleichungen übertragen werden, sodass dies keine Einschränkung darstellt. Zu beachten ist dabei, dass die Genauigkeit der numerischen Approximation maßgeblich durch die Wahl des zugrundeliegenden Gitters und des Zeitschrittverfahrens beeinflusst wird. Bekannte Symmetrieeigenschaften von Lösungen sollten sich im Gitter widerspiegeln.

6 Fazit und Aussicht

Aus den aufgeführten Ergebnissen können folgende Schlussfolgerungen gezogen werden. Die Implementierung der Finite-Elemente-Methode auf der GPU durch die Sprache CUDA brachte einen enormen Performanceanstieg bereits bei kleinen Modellen mit sich. Die Grafikkarte eines Computers ist demnach bei der Berechnung eines Zeitschrittes für die Finite-Elemente-Methode der CPU überlegen. Die Konstruktion der Systemmatrizen dauert im Vergleich zur Zeitschrittberechnung zu lange. Daher ist es empfehlenswert diesen Programmanteil ebenfalls zu parallelisieren. Es ist jedoch zu prüfen, ob dieser Schritt auf der GPU oder auf der CPU vorgenommen werden sollte. Bei der Konstruktion der Systemmatrizen handelt es sich um ein Reduktionsproblem, welches eine Reduzierung der Datenparallelität zur Folge hat. Demzufolge besteht die Möglichkeit, dass eine parallelisierte Implementierung auf der CPU dieses Problem durch eine gesteigerte Taktfrequenz und einen größeren Cache besser lösen könnte.

Für die Wahl der numerischen Methoden ist eine genaue Analyse der Problemstellung von Nutzen. Es gibt kein universelles Verfahren, welches alle Probleme gleichermaßen gut löst. Je nach Struktur der dünnbesetzten Systemmatrizen oder der partiellen Differentialgleichungen kann die Wahl einer Methode unterschiedlich geeignet sein. Auch hier stellt die Messung der Performance ein adäquates Mittel für die Auswahl der Verfahren dar. Für eine detailliertere Behandlung weiterer Verfahren zur Darstellung komplexerer Probleme sei hier auf [2, 3, 8, 10] verwiesen.

Um die Effizienz des gesamten Verfahrens in der Praxis zu bewerten, sollte dieses anhand realistischer physikalischer Problemstellungen, wie zum Beispiel der Strömungssimulation, gemessen und analysiert werden. Die Genauigkeit einer Voraussage hängt von der Diskretisierung des Berechnungsgebietes ab. Daher ist es ratsam, einen Algorithmus zur automatischen und adaptiven Gittergenerierung zu verwenden.

Eine Erweiterung des hier implementierten Verfahrens ist die Berechnung der Wel-

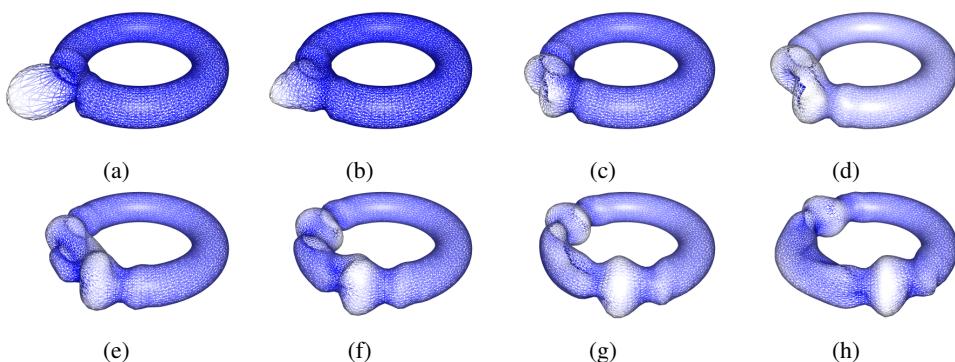


Abbildung 17: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf der gekrümmten Fläche eines Torus. Der Torus kann hier als eine Verallgemeinerung von periodischen Randbedingungen auf einem Quadrat angesehen werden.

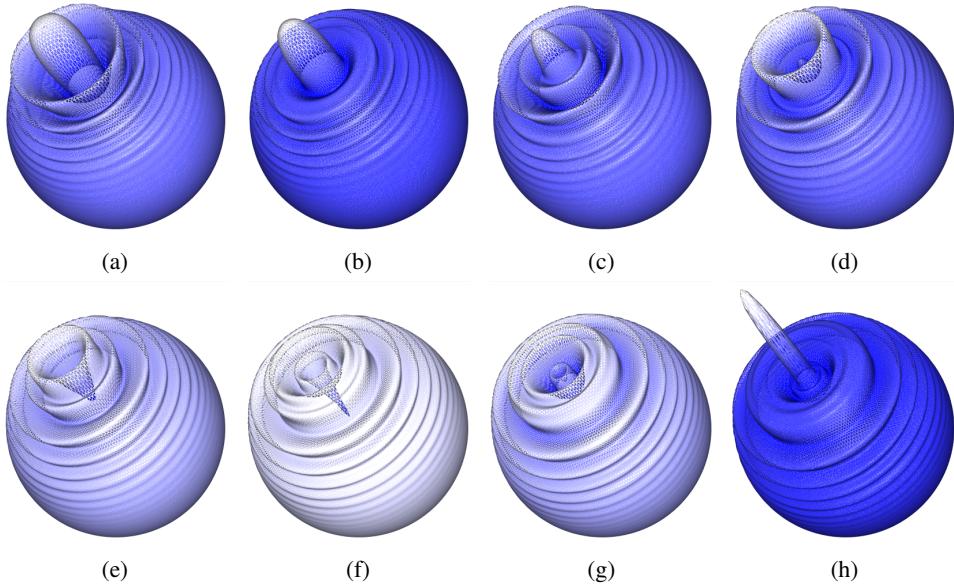


Abbildung 18: Die Abbildungen zeigen die Zeitevolution einer Simulation der Wellengleichung auf der gekrümmten Fläche einer Kugel. Durch die Geometrie der Kugel existieren keine Ränder und damit auch keine Randbedingungen.

lengleichung auf zweidimensionalen gekrümmten Oberflächen. In den Abbildungen 17 und 18 ist dies für einen Torus und eine Kugel veranschaulicht. Dies demonstriert, dass die Finite-Elemente-Methode auch auf mathematisch motivierte Probleme anwendbar ist.

Literatur

- [1] Alberty, J., C. Carstensen und S. A. Funken: *Remarks around 50 lines of MATLAB: Short Finite Element Implementation*. Numerical Algorithms, Feb 1998.
- [2] Bell, Nathan und Michael Garland: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report, Dezember 2008.
- [3] Bell, Nathan und Michael Garland: *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*. ACM, 2009.
- [4] Cheney, Ward und David Kincaid: *Numerical Mathematics and Computing*. Thomson, 6. Auflage, 2008, ISBN 978-0-495-11475-8.
- [5] Cohen, Gary C.: *Higher-Order Numerical Methods for Transient Wave Equations*. Springer, 2002, ISBN 978-3-642-07482-0.
- [6] CUDA, NVIDIA Developer Zone: *CUDA Toolkit Documentation*. <https://docs.nvidia.com/cuda/index.html>, 2018. [Online; accessed 30-August-2018].
- [7] Kirk, David B. und Wen mei W. Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010, ISBN 9780123814722.
- [8] Logan, Daryl L.: *A First Course in the Finite Element Method*. Thomson, 4. Auflage, 2007, ISBN 0-534-55298-6.
- [9] Meyers, Scott: *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 3. Auflage, 2008.
- [10] Nocedal, Jorge und Stephen J. Wright: *Numerical Optimization*. Springer, 2. Auflage, 2006, ISBN 978-0387-30303-1.
- [11] Patterson, David A. und John L. Hennessy: *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4. Auflage, 2011, ISBN 9780123747501.
- [12] Press, William H., Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2. Auflage, 2002, ISBN 0-521-43108-5.
- [13] Quarteroni, Alfio, Riccardo Sacco und Fausto Saleri: *Numerical Mathematics*. Springer, 2000, ISBN 0-387-98959-5.
- [14] Saad, Yousef: *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2. Auflage, 2003.

LITERATUR

- [15] Sanders, Jason und Edward Kandrot: *CUDA by Example: An Introduction to General-purpose GPU Programming*. Addison Wesley, 2011, ISBN 978-0-13-138768-3.
- [16] Schweizer, Ben: *Partielle Differentialgleichungen: Eine anwendungsorientierte Einführung*. Springer Spektrum, 2013.
- [17] Stroustrup, Bjarne: *The C++ Programming Language*. Addison Wesley, 4. Auflage, 2014, ISBN 978-0-321-95832-7.
- [18] Team, CUSP: *CUSP Documentation*. <https://cusplibrary.github.io/>, 2018. [Online; accessed 30-August-2018].
- [19] Team, Eigen: *Eigen Documentation*. <http://eigen.tuxfamily.org/dox/>, 2018. [Online; accessed 30-August-2018].

A Wärmeleitungsgleichung

Die einfachste zeitabhängige partielle Differentialgleichung ist die Wärmeleitungsgleichung [16, S. 175]. Bei ihr handelt es sich um eine parabolische partielle Differentialgleichung zweiter Ordnung, die ein einfaches Modell für die Berechnung instationärer Temperaturfelder bildet. Physikalisch basiert sie auf dem Energieerhaltungssatz.

Für die Einarbeitung eines Zeitparameters soll zunächst wieder ein räumliches Berechnungsgebiet gewählt werden, welches sich im Laufe der Zeit nicht verändert. Auf diesem lassen sich die räumlichen Ableitungen zu allen Zeiten analog zur Poisson-Gleichung betrachten. Um die Eindeutigkeit der Lösung auch nach dem Hinzufügen der Zeitableitungen zu gewährleisten, sind Anfangswerte von Nöten. Mit dieser Be trachtung lässt sich die Wärmeleitungsgleichung nun ohne neue Erkenntnisse analog zur Poisson-Gleichung definieren. [1, 16].

DEFINITION A.1: (Wärmeleitungsgleichung)

Es seien $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ ein Berechnungsgebiet und die folgenden Funktionen gegeben.

$$\begin{array}{ll} f \in L^2(\Omega \times [0, \infty)) & u_D \in H^1(\Omega \times [0, \infty)) \\ u_0 \in H^1(\Omega) & u_N \in L^2(\partial\Omega_N \times [0, \infty)) \end{array}$$

Eine Funktion $u \in H^1(\Omega \times [0, \infty))$ nennt man eine schwache Lösung der Wärmeleitungsgleichung, wenn die folgenden Gleichungen für alle $t \in [0, \infty)$ gelten.

$$\begin{array}{ll} \partial_t u - \Delta u = f & (\text{Wärmeleitungsgleichung}) \\ u(\cdot, 0) = u_0 & (\text{Anfangsbedingungen}) \\ u(\cdot, t)|_{\partial\Omega_D} = u_D(\cdot, t)|_{\partial\Omega_D} & (\text{Dirichlet-Randbedingungen}) \\ \langle \nabla u(\cdot, t)|_{\partial\Omega_N}, \nu \rangle = u_N(\cdot, t) & (\text{Neumann-Randbedingungen}) \end{array}$$

Das Tupel $([\Omega], f, u_0, u_D, u_N, u)$ wird dann auch Wärmeleitungsproblem genannt.

Ebenfalls analog zur Poisson-Gleichung ist es auch möglich, die Wärmeleitungs gleichung in eine schwache Formulierung zu transformieren. Auch bei diesem Vorgang entstehen keine Überraschungen bei der mathematischen Vorgehensweise. Zu bemerken sei hier, dass für die Konstruktion der Finite-Elemente-Methode die rigorose schwache Formulierung der Wärmeleitungsgleichung keine direkte Rolle spielt. Beim Übergang zur Diskretisierung erweist es sich als praktisches Vorgehen, jegliche Zeitableitung zuvor durch einen diskreten Differenzenquotienten zu ersetzen, sodass es

sich bei dem resultierenden Gleichungssystem streng genommen nicht mehr um eine zeitabhängige Differentialgleichung handelt. [1, 16]

Für die zeitliche Diskretisierung der Wärmeleitungsgleichung kann man die sogenannte Rothe-Methode, wie sie in [16, S. 211 ff] und [1] beschrieben ist, verwenden. Sie basiert auf dem impliziten Euler-Verfahren und verhindert damit die Divergenz numerischer Lösungen [13, S. 472 f]. Auch andere Zeitschrittverfahren, wie die Crank-Nicolson-Methode oder die Familie der Runge-Kutta-Verfahren sind jedoch denkbar [4, 13].

Es seien nun ein kleiner Zeitschritt $dt \in (0, \infty)$, ein Berechnungsgebiet $[\Omega] := (\Omega, \partial\Omega_D, \partial\Omega_N, \nu)$ und ein Wärmeleitungsproblem $([\Omega], f, u_0, u^{(D)}, u^{(N)}, u)$ gegeben. Durch dt wird das Zeitintervall $T := [0, \infty)$ diskretisiert und durch die Menge $[T] := \{k \cdot dt \mid k \in \mathbb{N}_0\}$ dargestellt. Weiterhin bezeichne eine natürliche Zahl $n \in \mathbb{N}$ als Index der zeitabhängigen Funktionen $f, u^{(D)}, u^{(N)}$ und u deren Diskretisierung zum Zeitpunkt $n \cdot dt$. Dieses Schema soll am Beispiel von u demonstriert werden.

$$u \longleftrightarrow (u_n)_{n \in \mathbb{N}_0}, \quad u(\cdot, ndt) \longleftrightarrow u_n$$

Zu beachten ist hier, dass die Diskretisierung u_n im Allgemeinen $u(\cdot, ndt)$ nur annähert und nicht exakt beschreibt. Für die Diskretisierung der zeitlichen Ableitung folgt durch die Anwendung des impliziten Euler-Verfahrens ein ähnliches Schema.

$$\partial_t u \longleftrightarrow (\partial_t u_n)_{n \in \mathbb{N}}, \quad \partial_t u(\cdot, ndt) \longleftrightarrow \frac{u_n - u_{n-1}}{dt}$$

Mit diesen Transformationen ist es jetzt möglich, die gesamte Wärmeleitungsgleichung zu diskretisieren, wie im folgenden Schema gezeigt.

$$\partial_t u - \Delta u = f \quad \longleftrightarrow \quad \frac{u_n - u_{n-1}}{dt} - \Delta u_n = f_n$$

Bei der Berechnung wird davon ausgegangen, dass u_{n-1} entweder als Anfangswert u_0 gegeben ist oder durch einen vorherigen Zeitschritt berechnet wurde. Die einzige Unbekannte in der diskretisierten Gleichung ist damit u_n , für die man den folgenden Ausdruck erhält.

$$(id - dt\Delta) u_n = dt f_n + u_{n-1}$$

Für jeden Zeitpunkt $t \in [T]$ handelt es sich hier um eine zeitunabhängige partielle Differentialgleichung zweiter Ordnung, deren schwache Lösung analog zu der der Poisson-Gleichung konstruiert werden kann. Zunächst werden die Dirichlet-Randbedingungen wieder über eine Transformation in die Gleichung eingearbeitet.

$$s_n \in H_D^1(\Omega), \quad s_n := u_n - u_n^{(D)}$$

Durch die Integration mit Testfunktionen und die Anwendung des Gaußschen Satzes erhält man nun für die schwachen Lösungen der Zeit-diskretisierten Wärmeleitungsgleichung die folgende etwas längliche Formulierung für alle $\varphi \in H_D^1(\Omega)$.

$$\begin{aligned} & \int_{\Omega} s_n \varphi \, d\lambda + dt \int_{\Omega} \langle \nabla s_n, \nabla \varphi \rangle \, d\lambda \\ &= dt \left[\int_{\Omega} f_n \varphi \, d\lambda + \int_{\partial\Omega_N} u_n^{(N)} \varphi \, d\sigma - \int_{\Omega} \langle \nabla u_n^{(D)}, \nabla \varphi \rangle \, d\lambda \right] \\ &+ \int_{\Omega} u_{n-1} \varphi \, d\lambda - \int_{\Omega} u_n^{(D)} \varphi \, d\lambda \end{aligned}$$

B ELLPACK-Format

Ein weiteres Speicherformat, welches sehr gut für Vektorarchitekturen geeignet ist, wird durch das sogenannte ELLPACK-Format (ELL) beschrieben [2]. Für eine Matrix $M \in \mathbb{R}^{n \times n}$ mit maximal $k \in \mathbb{N}$, $k \leq n$ Einträgen in jeder Zeile, die nicht Null sind, speichert ELL diese Werte in einem Tupel $[M]_{\text{ELL}}^{(k)}$ zweier $n \times k$ -Matrizen.

$$[M]_{\text{ELL}}^{(k)} := (D, C), \quad D \in \mathbb{R}^{n \times k}, \quad C \in \{i \in \mathbb{N}_0 \mid i < n\}^{n \times k}$$

D speichert dabei alle Werte ungleich Null, indem es diese so weit wie möglich innerhalb einer Zeile nach links bewegt. Bei Zeilen, die mehr als $n - k$ Nullen beinhalten, werden überschüssige Einträge mit Nullen aufgefüllt. Die Einträge der zweiten Matrix C beschreiben den jeweiligen Spaltenindex eines Elementes ungleich Null. Hier werden bei Zeilen, die mehr als $n - k$ Nullen beinhalten, überschüssige Einträge durch einen festgelegten Sentinel aufgefüllt. Dieses Vorgehen soll durch das folgende Beispiel veranschaulicht werden. [2, 3, 12]

$$M = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}, \quad [M]_{\text{ELL}}^{(3)} = \left(\begin{pmatrix} 3 & 0 & 0 \\ 22 & 17 & 0 \\ 7 & 5 & 1 \\ 0 & 0 & 0 \\ 14 & 8 & 0 \end{pmatrix}, \begin{pmatrix} 1 & * & * \\ 0 & 4 & * \\ 0 & 1 & 3 \\ * & * & * \\ 2 & 4 & * \end{pmatrix} \right)$$

Unterscheidet sich die maximale Anzahl von Einträgen ungleich Null in einer Zeile nicht substantiell von der durchschnittlichen Anzahl von Einträgen ungleich Null einer Zeile, so stellt ELL eine sehr effiziente Sparse-Matrix-Datenstruktur dar. In der Finite-Elemente-Methode würde demnach die Performance des Verfahrens von der Struktur des Gitters abhängen. [2]

Eigenständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Quellen angefertigt habe. Die eingereichte Arbeit ist nicht anderweitig als Prüfungsleistung verwendet worden oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen.

Seitens des Verfassers bestehen keine Einwände, die vorliegende Bachelorarbeit für die öffentliche Benutzung zur Verfügung zu stellen.

Jena, 31. August 2018

Markus Pawellek