FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Algorithmische Umsetzung des Oliker-Prussner-Verfahrens

B A C H E L O R A R B E I T

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang Mathematik

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Fakultät für Mathematik und Informatik

eingereicht von Yannick Höffner

geb. am 23.07.2003 in Eisenach

Betreuer: Prof. Dr. D. Gallistl

Jena, Februar 2025

Abstract

Die Lösung partieller Differentialgleichungen ist eine der zentralen Herausforderungen der modernen Mathematik und in sämtlichen Naturwissenschaften von wesentlichem Interesse. Die rein analytische und exakte Lösung dieser Gleichungen ist aufgrund ihrer Komplexität i.d.R. nicht möglich, sodass wir numerische Verfahren entwickeln müssen, um diese näherungsweise zu lösen. Im Rahmen der vorliegenden Bachelorarbeit haben wir das Oliker-Prussner-Verfahren [1] implementiert und können damit eine voll nichtlineare partielle Differentialgleichung, die zweidimensionale Monge-Ampère-Gleichung, lösen. Der Schwerpunkt dieser Arbeit lag in der Erstellung einer Software, welche dieses 36 Jahre alte, aber dennoch selten verwendete Verfahren softwaretechnisch umsetzt. Dabei mussten wir teilweise eigene Teilalgorithmen entwickeln, welche die mathematische Formulierung von Oliker und Prussner für einen Computer greifbar machen. Des Weiteren haben wir anhand numerischer Experimente die Verhaltensweise des Verfahrens intensiv untersucht und Anhaltspunkte für zukünftige Forschung gefunden.

ACHTUNG

Dieses hier veröffentlichte PDF-Dokument entspricht nicht der tatsächlichen, originalen Bachelorarbeit. Bei diesem Dokument handelt es sich um eine korrigierte und an die GitHub-Veröffentlichung angepasste Fassung. Aufgrund eines Softwarefehlers enthielt das originale Kapitel 4 ein fehlerhaftes Beispiel. Dieses Beispiel sowie der Verweis darauf im Fazit wurden in dieser Datei entfernt, damit diese fehlerhaften Aussagen nicht verbreitet werden.
Da dieses Dokument Bestandteil eines GitHub-Repositories ist und dieses somit eine `README.md`-Datei enthält, entfällt der Nutzen des Anhangs A, da dieser in aktualisierter Form in die `README.md`-Datei ausgelagert wurde.

HINWEIS

Die hierbei entwickelte Software ist quelloffen und auf GitHub öffentlich einsehbar unter dem Link: https://github.com/yannick-hoeffner/oliker_prussner_method

Inhaltsverzeichnis

1. Einleitung

Partielle Differentialgleichungen sind eines der wichtigsten Werkzeuge der modernen Mathematik. Sie ermöglichen es, komplizierte naturwissenschaftliche Sachverhalte zu beschreiben und zu modellieren. Somit sind etwa die Maxwell-Gleichungen die Grundlage des Elektromagnetismus [2], die Schrödingergleichung bildet eine der grundlegenden Gleichungen der Quantenmechanik [3], die Navier-Stockes-Gleichungen modellieren die Strömungsmechanik [4] und die Black-Scholes-Gleichung bildet das Fundament der modernen Finanzmärkte [5]. Daraus ergibt sich ein enormes Interesse an der Lösung partieller Differentialgleichungen, welches sich über das gesamte wissenschaftliche Spektrum verteilt. Es ist also kaum verwunderlich, dass sämtliche moderne (physikalische) Simulationssoftware intern auf Algorithmen zum Lösen partieller Differentialgleichungen basiert.

In dieser Bachelorarbeit wollen wir uns mit dem Oliker-Prussner-Verfahren beschäftigen. Das 1989 von Vladimir I. Oliker und Laird D. Prussner veröffentlichte Verfahren beschreibt einen Algorithmus zum Lösen der Monge-Ampère-Gleichung in zwei Dimensionen [1]. Die nach Gaspard Monge und André-Marie Ampère benannte partielle Differentialgleichung entsprang ursprünglich dem Problem des optimalen Transports, also der Frage, wie man eine Dichtefunktion möglichst kosteneffizient in eine andere transformieren kann [6, 7, Abschnitt 4.6]. Allerdings taucht die Monge-Ampère-Gleichung als führender Term in vielen weiteren Problemen auf, so beispielsweise in der numerischen Wettervorhersage [8, 7, Abschnitt 4.9] oder dem Reflektor Design Problem, bei dem die Form eines Reflektors berechnet werden soll, der ein bestimmtes Muster projiziert [9, 10]. Aber auch das Minkowski Problem der Gaußschen Krümmung lässt sich im Kern auf die Monge-Ampère-Gleichung zurückführen [7, Abschnitt 2.6].

Obwohl die Monge-Ampère-Gleichung auf den ersten Blick eine sehr simple Gestalt hat, so ist ihre Lösung aufgrund der vollen Nichtlinearität höchst anspruchsvoll. Umso beeindruckender ist die Leistung von Oliker und Prussner, ein konvergentes Verfahren zu entwickeln, welches einen zweidimensionalen Fall der Monge-Ampère-Gleichung zuverlässig löst. Allerdings ist dieses Verfahren aufgrund seiner geometrischen Bauart schwierig zu implementieren und voraussichtlich langsam in der Ausführung. Und obwohl das Verfahren als theoretisches Ergebnis häufig zitiert wird und von einigen Autoren numerische Experimente durchgeführt wurden, liegen gegenwärtig kaum Informationen über eine tat-

sächliche Implementierung dieses Verfahrens in seiner Originalfassung vor. Das Hauptziel dieser Arbeit ist deshalb eine funktionsfähige Software zu entwickeln, welche das originale Verfahren von Oliker und Prussner aus [1] implementiert. Die Bachelorarbeit soll ferner auch als eine Art Implementierungsanleitung bzw. Dokumentation zur Funktionsweise der Software dienen. Somit wird es eine große Inhaltliche Überschneidung mit dem Paper [1] von Oliker und Prussner geben.

Diese Arbeit gliedert sich wie folgt. Im zweiten Kapitel wird die Monge-Ampère-Gleichung von theoretischer Seite her eingeführt und die Herleitung einer schwachen Formulierung sowie einer Diskretisierung der Gleichung im numerischen Sinne beschrieben. Im dritten Kapitel wird dann das Oliker-Prussner-Verfahren vorgestellt sowie unsere Implementierung dessen ausführlich beschreiben. Anschließend erfolgt im vierten Kapitel einerseits ein Nachweis für die Funktionsfähigkeit unserer Implementierung und andererseits einige numerische Experimente zum Verhalten des Verfahrens. Ein Fazit soll abschließend einen Ausblick auf weiterführende Möglichkeiten zu Arbeiten an dieser Software bieten.

Aufgrund des Umfangs dieser Aufgabe schließt diese Bachelorarbeit an einen Seminarvortrag an, in welchem wir einen stark vereinfachter Prototyp des Verfahrens implementiert haben. Dieser Prototyp war nur in der Lage die Monge-Ampère-Gleichung für die Normalparabel zu lösen und setzte das Verfahren nur teilweise um. Im Rahmen der Bachelorarbeit wurde dieser Protoyp ausgebaut und zu einer vollumfänglichen Implementierung des Oliker-Prussner-Verfahrens vervollständigt, welche nun die Gleichung zuverlässig für sämtliche Eingaben löst.

Einen besonderen Dank möchte ich an Lukas Gehring richten, dessen konstruktive Beratung und theoretisches Fachwissen maßgeblich zum Gelingen dieser Arbeit beigetragen haben. Ferner möchte ich mich bei meinem Betreuer Prof. Dr. Dietmar Gallistl bedanken, für dieses anspruchsvolle Thema und die Betreuung des (inklusive Vorarbeit) mittlerweile fast einjährigen Arbeitsprozesses.

2. Die Monge-Ampère-Gleichung

In diesem Kapitel stellen wir die grundlegende Gleichung vor, welche das Oliker-Prussner-Verfahren lösen soll: Die Monge-Ampère-Gleichung.

Dabei wollen wir eine schwache Form der Gleichung herleiten, die Alexandrov-Formulierung, und einige theoretische Ergebnisse hinsichtlich der Lösbarkeit erwähnen. Anschließend wollen wir die schwache Formulierung diskretisieren, damit wir uns in einem geeigneten Setting befinden, in welchem wir im folgenden Kapitel das Oliker-Prussner-Verfahren entwickeln können. [1]

2.1. Die klassische Formulierung

Für das Oliker-Prussner-Verfahren betrachten wir lediglich einen Spezialfall der zweidimensionalen Monge-Ampère-Gleichung, welche eine voll nichtlineare, degenerierte, partielle Differentialgleichung zweiter Ordnung mit Dirichlet Randwerten ist:

Definition 2.1.1: Sei $\Omega \subset \mathbb{R}^2$ ein beschränktes und konvexes Gebiet, $f : \Omega \to [0, \infty)$ eine nichtnegative Funktion auf Ω und $g : \partial\Omega \to \mathbb{R}$ eine Funktion auf dem Rand von Ω. Dann ist die *Monge-Ampère-Gleichung* gegeben durch

$$\det\left(D^2 u\right) = \frac{\partial^2 u}{\partial x^2}\frac{\partial^2 u}{\partial y^2} - \left(\frac{\partial^2 u}{\partial x \partial y}\right)^2 = f \quad \text{in } \Omega \text{ und} \tag{2.1a}$$

$$u = g \quad \text{auf } \partial\Omega, \tag{2.1b}$$

wobei $D^2 u$ die Hesse-Matrix von u ist.
Dann heißt $u \in C^2(\Omega) \cap C\left(\bar{\Omega}\right)$ *klassische Lösung* der Monge-Ampère-Gleichung wenn sie 2.1 punktweise in $\bar{\Omega}$ erfüllt.

Die Monge-Ampère-Gleichung folgt somit direkt aus der Determinante der Hessematrix und wird i.A. auch so für höhere Dimensionen definiert. Da wir insbesondere

[1] Die Abschnitte 2.1 und 2.2 sind im wesentlichen eine Zusammenfassung von [7, Kapitel 1 bis 3], und der Abschnitt 2.3 ist eine Erläuterung der Diskretisierung wie sie Oliker und Prussner selbst in [1] verwendeten.

$\det(D^2u) = f \geq 0$ fordern, ist es nicht möglich, dass die Eigenwerte der Hesse-Matrix ein unterschiedliches Vorzeichen haben[2]. Sie haben alle das gleiche Vorzeichen oder sind Null. Folglich ist jede klassische Lösung eine konkave oder konvexe Funktion (ist sogar $f > 0$, dann ist u strikt konkav oder strikt konvex). Da man jede konkave Funktion durch Negieren der Funktionswerte in eine konvexe Funktion umwandeln kann, genügt es, wenn wir uns mit konvexen Lösungen befassen. Das bedeutet, dass für eine klassische Lösung u für alle Punkte $a_1, a_2 \in \Omega$ und $\lambda \in [0,1]$ gilt, dass

$$u(\lambda a_1 + (1-\lambda)a_2) \leq \lambda u(a_1) + (1-\lambda)u(a_2). \tag{2.2}$$

Man kann nun zeigen, dass solche klassischen Lösungen $u \in C^{k+2,\alpha}(\bar{\Omega})$, für $k \geq 2$ und $\alpha \in (0,1)$, unter strengen Regularitätsanforderungen an f, g und $\partial\Omega$ eindeutig existieren (siehe [7, Kapitel 3.1] für weitere Informationen). Die theoretische Betrachtung der klassischen Monge-Ampère-Gleichung werden wir an dieser Stelle nicht weiter vertiefen, da wir uns für den Rest dieser Arbeit lediglich mit der schwachen Formulierung beschäftigen werden.

2.2. Die Alexandrov-Formulierung

Betrachten wir eine klassische Lösung u der Monge-Ampère-Gleichung nach Definition 2.1.1. Da u eine reellwertige Funktion ist, ist die Hesse-Matrix von u gleich der Jakobi-Matrix vom Gradienten ∇u. Damit erhalten wir für jede Borelmenge $\omega \subset \Omega$ nach dem Transformationssatz für mehrdimensionale Integrale[3]

$$\int_\omega \det\left(D^2u(x,y)\right) \mathrm{d}x\mathrm{d}y = \int_\omega 1 \left|\det\left(D\nabla u(x,y)\right)\right| \mathrm{d}x\mathrm{d}y = \int_{\nabla u(\omega)} 1 \mathrm{d}p\mathrm{d}q. \tag{2.3}$$

Dabei ist die rechte Seite der Gleichung 2.3 nichts anderes als das Lebesguemaß λ der Bildmenge $\nabla u(\omega)$. Integrieren wir die Monge-Ampère-Gleichung 2.1 über ω, so erhalten wir für integrierbare f die Maßgleichung

$$\lambda(\nabla u(\omega)) = \int_\omega f(x,y)\mathrm{d}x\mathrm{d}y. \tag{2.4}$$

Jede klassische Lösung erfüllt also die Maßgleichung 2.4 für jede Borelmenge $\omega \subset \Omega$. Gleichzeitig öffnet diese Maßformulierung den Raum der Lösungsfunktionen, da eine Lö-

[2] Man beachte, dass wir hier aufgrund der Einschränkung auf 2 Dimensionen auch nur 2 Eigenwerte haben.

[3] Um die Anwendung des Transformationssatzes zu verdeutlichen, führen wir einen notationellen Variablenwechsel vom xy-Koordinatensystem zum pq-Koordinatensystem durch.

sung u der Maßgleichung 2.4 nicht mehr notwendigerweise zweimal differenzierbar sein muss, es genügt die Wohldefiniertheit von ∇u. Aber auch die Wohldefiniertheit von ∇u kann gewissermaßen als restriktiv betrachtet werden. Gerade in der Numerik werden nämlich häufig stückweise lineare Funktionen betrachtet, also solche Funktionen, die an manchen Stellen gar nicht erst differenzierbar sind. Dennoch wollen wir an der geometrischen Idee des Gradienten festhalten. Für einen Punkt $a \in \Omega$ gibt der Gradient an, welche Steigung die Tangentialebene am Graphen von u am Punkt $(a, u(a))$ hat. Um diese geometrische Interpretation zu erhalten und gleichzeitig den Raum der Lösungsfunktion für nicht differenzierbare Funktionen zu öffnen, können wir den Gradienten durch den Subgradienten ersetzen:

Definition 2.2.1: Sei Ω eine konvexe Menge und $u : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ eine konvexe Funktion. Dann ist der *Subgradient* von u an der Stelle $(x, y) \in \Omega$ definiert durch

$$\partial u(x, y) := \left\{(p, q) \in \mathbb{R}^2 | \forall (\tilde{x}, \tilde{y}) \in \mathbb{R}^2 : u(x, y) + p(x - \tilde{x}) + q(y - \tilde{y}) \leq u(\tilde{x}, \tilde{y})\right\}. \tag{2.5}$$

Für eine Borelmenge $\omega \subset \Omega$ definieren wir $\partial u(\omega) := \cup_{(x,y)\in\omega} \partial u(x, y)$.

Geometrisch gesprochen ist der Subgradient von u an der Stelle $a \in \Omega$ die Menge der Steigungen **aller** Stützebenen[4], welche am Graphen von u am Punkt $(a, u(a))$ anliegen. Dabei beschreibt p die Steigung in x-Richtung und q in y-Richtung. Das Besondere am Subgradienten ist, dass er für alle (nicht notwendigerweise strikt) konvexen Funktionen überall im Inneren von Ω wohldefiniert ist.

Ferner kann man zeigen, dass $\lambda(\partial u(\cdot))$ ein lokal endliches Borelmaß auf Ω ist [7, Kapitel 2]. Ersetzen wir also in der Maßgleichung 2.4 den Gradienten durch den Subgradienten (und ersetzen im gleichen Atemzug das Integral der rechten Seite notationell durch das Maß μ), so erhalten wir die Alexandrov-Formulierung:

Definition 2.2.2: Sei μ ein endliches Borelmaß auf Ω und $g : \partial\Omega \to \mathbb{R}$ eine skalar Funktion auf dem Rand von Ω, dann ist die *Alexandrov-Formulierung* der Monge-Ampère-Gleichung gegeben durch

$$\lambda(\partial u(\omega)) = \mu(\omega) \qquad \forall \omega \subset \Omega \text{ und} \tag{2.6a}$$

$$u = g \qquad \text{auf } \partial\Omega. \tag{2.6b}$$

Ferner heißt $u \in C(\bar{\Omega})$ *Alexandrov-Lösung* der Monge-Ampère-Gleichung, wenn sie 2.6 für alle Borelmengen $\omega \subset \Omega$ erfüllt. $\lambda(\partial u(\cdot))$ wird auch als *Monge-Ampère-Maß* bezeichnet.

[4] Eine Stützebene des Graphen S_u von u ist eine Ebene E welche S_u schneidet, also $E \cap S_u \neq \varnothing$ und gleichzeitig den $\mathbb{R}^3$ so in zwei Halbräume teilt, dass S_u vollständig im Abschluss von einem dieser Halbräume enthalten ist.

Man kann nun zeigen, dass für strikt konvexe Gebiete Ω und stetige $g \in C(\partial\Omega)$ die Alexandrov-Lösung eindeutig existiert [7, Theorem 2.14]. Zusätzlich schafft die Verallgemeinerung des Integrals $\int_\omega f(x,y)\mathrm{d}x\mathrm{d}y$ zu einem beliebigen Maß μ einen theoretischen Rahmen, um auch Diracmaße u.ä. zu betrachten.

Nun wollen wir diese schwache Formulierung diskretisieren und somit in ein endlichdimensionales Setting umwandeln, aus welchem wir dann das Oliker-Prussner-Verfahren entwickeln können.

2.3. Diskretisierung der Alexandrov-Formulierung

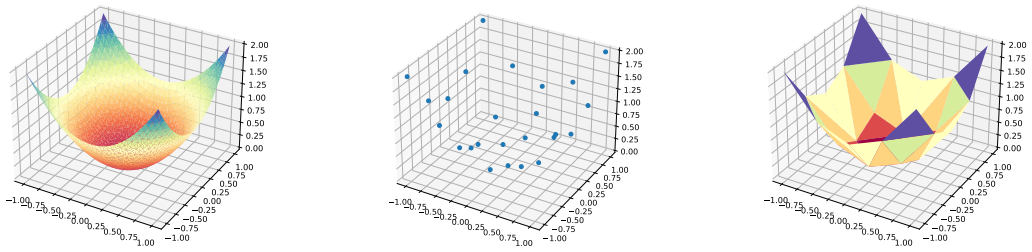
Um das unendlich dimensionale Problem 2.2.2 in einen endlich dimensionalen Rahmen zu packen, beginnen wir damit die Menge Ω zu diskretisieren. Dabei betrachten wir für Ω nur konvexe, beschränkte Gebiete, deren Rand ein geschlossener Polygonzug Γ mit endlich vielen Eckpunkten $b_1, \ldots, b_N$ ist.

Die stetige Funktion $g : \Gamma \to \mathbb{R}$, welche die Randbedingung beschreibt, betrachten wir als stückweise affin. Somit ist sie durch die Werte an den Eckpunkten b_i eindeutig definiert.

Das Innere der Menge Ω diskretisieren wir mit M beliebig gewählten, verschiedenen Punkten $a_1, \ldots, a_M$ (Notation: $b_{N+j} := a_j$). Betrachten wir nun eine auf Ω definierte konvexe Funktion u, so hat auch die Menge der Funktionswerte $\mathcal{M} := \{u(b_i) | i = 1, \ldots, M+N\}$ eine „konvexe Struktur“ (vgl. Abb. 2.1a & 2.1b). Nun liefert der in z-Richtung betrachtete, untere Rand der konvexen Hülle von $\mathcal{M}$ einen stückweise linearen Funktionsgraphen welcher die Menge $\mathcal{M}$ interpoliert (vgl. Abb. 2.1c). Wir bezeichnen die so konstruierte diskretisierte Form der Funktion u als u_h und bezeichnen den Graphen mit S_{u_h}.

Diese Konstruktion hat 3 wesentliche Vorteile. Erstens ist die Interpolationsfunktion u_h durch die Eindeutigkeit der konvexen Hülle eindeutig definiert. Zweitens ist die Funktion u_h auf dem Rand Γ stückweise linear und stimmt mit g überein, sofern $u(b_i) = g(b_i)$ für alle $i = 1, \ldots, N$ gilt. Und drittens ist der Subgradient $\partial u_h(x,y)$ für alle Punkte $(x,y) \in \Omega$ wohldefiniert und das Monge-Ampère-Maß ist höchstens an den Stellen $a_i, i = 1, \ldots, M$ ungleich 0.

Um uns vom dritten Punkt zu überzeugen, betrachten wir einen beliebigen inneren Punkt $(x,y) \in \Omega$. Befindet sich $(x, y, u_h(x,y))$ im Inneren einer Fläche vom Graphen S_{u_h} (vgl. Punkt 1 auf Abb. 2.2), so ist die Stützebene durch diese Fläche eindeutig definiert. Die Menge $\partial u_h(x,y)$ besteht somit nur aus einem Punkt und hat das Lebesquemaß 0. Befindet sich $(x, y, u_h(x,y))$ auf einer Kante zwischen zwei Flächen vom Graphen S_{u_h} (vgl. Punkt 2 auf Abb. 2.2), so liegt der Anstieg einer Stützebene zwischen den



(a) **(b)** **(c)**

Abbildung 2.1.: Konstruktion von u_h anhand des Beispiels $u(x, y) = x^2 + y^2$ auf dem Gebiet $\Omega = [-1, 1]^2$. **(a)** Graph der Funktion u **(b)** Punktwolke $\mathcal{M}$ für 25 äquidistante Punkte $b_i \in \Omega$ **(c)** Graph der Funktion u_h als unterer Rand der konvexen Hülle von $\mathcal{M}$

Anstiegen dieser zwei Flächen. Die Menge $\partial u_h(x, y)$ besteht somit aus einer Linie und hat demnach ebenfalls das Lebesquemaß 0 (man beachte, dass wir das zwei dimensionale Lebesquemaß nutzen). Damit muss der Punkt $(x, y, u_h(x, y))$ also ein echter Eckpunkt des Graphen S_{u_h} sein, damit der Subgradient überhaupt ein Lebesquemaß ungleich 0 haben kann (vgl. Punkt 3 auf Abb. 2.2)). Per Konstruktion über die konvexe Hülle sind diese Eckpunkte aber höchstens die Punkte $(a_i, u(a_i))$ für $i = 1, \ldots, M$.

Dann können wir den Raum W_M der infrage kommenden Lösungsfunktionen u_h wie folgt definieren:

Definition 2.3.1: Sei $\Omega \subset \mathbb{R}^2$ ein konvexes, beschränktes Gebiet wobei der Rand Γ ein geschlossener Polygonzug mit endlich vielen Eckpunkten $b_1, \ldots, b_N$ ist. Sei $g : \Gamma \to \mathbb{R}$ eine stetige, stückweise lineare Funktion auf dem Rand, welche auf jeder Kante $[b_i, b_{i+1}]$ des Randes linear ist. Ferner seien $a_1, \ldots, a_M \in \Omega$ verschiedene, innere Punkte.
Wir definieren den *Lösungsraum* W_M als Raum aller konvexen, stückweise linearen, stetigen Funktionen $w : \Omega \to \mathbb{R}$ mit

(i) $w(x, y) = g(x, y)$ für alle Randpunkte $(x, y) \in \Gamma$ und

(ii) das Monge-Ampère-Maß ist im Inneren von Ω höchstens an den Punkten a_i für $i = 1, \ldots, M$ ungleich 0.

Nachdem wir nun einen diskreten Lösungsraum haben, müssen wir noch die Alexandrov-Formulierung (Definition 2.2.2) auf diesen diskreten Raum übertragen.

Aufgrund der Eigenschaft (ii) von Definition 2.3.1 sind nur die Gitterpunkte a_i wichtig für die Berechnung des Subgradienten. Für $u_h \in W_M$ folgt also für eine beliebige Borelmenge $\omega \subset \Omega$

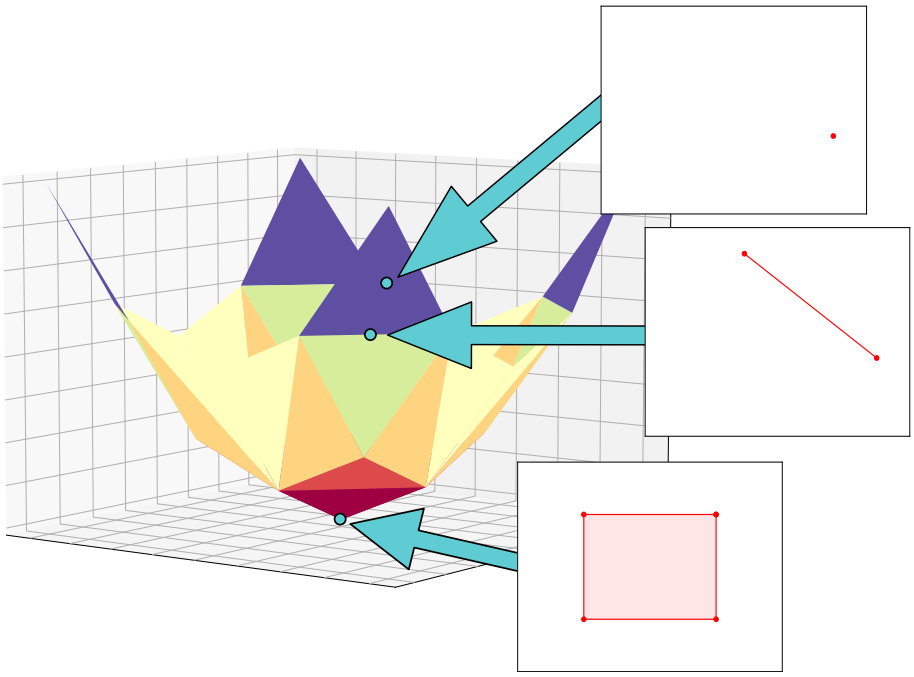


Abbildung 2.2.: Beispielhafter Subgradient der Funktion u_h aus Abb. 2.1 an drei verschiedenen Punkten. Punkt 1 (oben) liegt auf einer Fläche von S_{u_h}, Punkt 2 (mittig) liegt auf einer Kante von S_{u_h} und Punkt 3 (unten) ist ein Eckpunkt von S_{u_h}.

$$\lambda(\partial u_h(\omega)) = \lambda(\cup_{a_i \in \omega} \partial u_h(a_i)) = \sum_{a_i \in \omega} \lambda(\partial u_h(a_i)). \tag{2.7}$$

Damit vereinfacht sich Gleichung 2.6a der Alexandrov Formulierung zu

$$\sum_{a_i \in \omega} \lambda(\partial u_h(a_i)) = \mu(\omega). \tag{2.8}$$

Nun müssen wir das Maß μ diskretisieren zu einem Maß, welches ausschließlich an den Punkten a_i ungleich 0 ist, um Gleichung 2.8 vollständig zu diskretisieren. Ursprünglich wurde das Maß μ als Lebesquemaß mit Dichtefunktion f eingeführt, was eine Diskretisierung sehr flexibel macht. Sei dafür $\Omega_1, \ldots, \Omega_M$ eine disjunkte Partition von Ω mit der Eigenschaft, dass Ω_i ausschließlich den Punkt a_i und eine Umgebung dieses Punktes enthält. Damit meinen wir insbesondere, dass $a_i \notin \Omega_j$ und $\Omega_i \cap \Omega_j = \varnothing$ für $i \neq j$ und

dass $\Omega = \bigcup_{j=1}^{M} \Omega_j$. Dann kann man μ wie folgt diskretisieren

$$\mu(a_i) := \int_{\Omega_i} f(x)\mathrm{d}x. \tag{2.9}$$

Es ist wichtig anzumerken, dass es keine kanonische a priori Wahl dieser Partition gibt! Allerdings hat die Wahl der Partition einen sehr großen Einfluss darauf wie genau die Monge-Ampère-Gleichung 2.1 im diskreten Sinne modelliert wird und hat dementsprechend einen großen Einfluss auf die Lösungsfunktion, welche das Oliker-Prussner-Verfahren berechnet.

Damit verändert sich Gleichung 2.8 zu

$$\sum_{a_i \in \omega} \lambda(\partial u_h(a_i)) = \sum_{a_i \in \omega} \mu(a_i) := \sum_{a_i \in \omega} \int_{\Omega_i} f(x)\mathrm{d}x. \tag{2.10}$$

Da Gleichung 2.10 insbesondere auch für alle Borelmengen ω gilt, welche lediglich einen der Punkte a_i enthalten, bekommen wir somit M individuelle Gleichungen für jeden einzelnen inneren Punkt a_i. Damit können wir nun die diskrete Alexandrov-Formulierung der Monge-Ampère-Gleichung formulieren:

Definition 2.3.2: Seien Ω und g wie in Definition 2.3.1 diskretisiert. Ferner sei μ ein endliches Maß auf Ω, welches ausschließlich an den Stellen a_i ungleich 0 ist. Dann ist die *diskrete Alexandrov-Formulierung* der Monge-Ampère-Gleichung gegeben durch:

$$\lambda(\partial u_h(a_i)) = \mu(a_i) \qquad \forall i = 1, \ldots, M \tag{2.11a}$$
$$u(b_i) = g(b_i) \qquad \forall i = 1, \ldots, N \tag{2.11b}$$

$u_h \in W_M$ heißt *diskrete Alexandrov-Lösung*, wenn sie Gleichung 2.11 für alle inneren Punkte a_i und alle Randpunkte b_i erfüllt

Für die Existenz und die Eindeutigkeit der Lösungen der diskreten Alexandrov-Formulierung kann man das Oliker-Prussner-Verfahren als konstruktiven Beweis ansehen [1].

3. Das Oliker-Prussner-Verfahren

In diesem Kapitel werden wir basierend auf der diskreten Alexandrov-Formulierung aus Definition 2.3.2 das Oliker-Prussner-Verfahren erarbeiten.

Dabei wollen wir zunächst die grobe Grundidee von Oliker und Prussner vorstellen. Anschließend wollen wir diese Grundidee Stück für Stück vertiefen und detailliert beschreiben wie wir die einzelnen Teile implementiert haben. Sofern wir es nicht explizit anders beschreiben, entstammt sämtliche theoretische Arbeit zu diesem Verfahren aus der Originalveröffentlichung von Oliker und Prussner [1], lediglich die konkrete Wahl der Implementierung entstammt unserer Ausarbeitung.

3.1. Grundidee von Oliker und Prussner

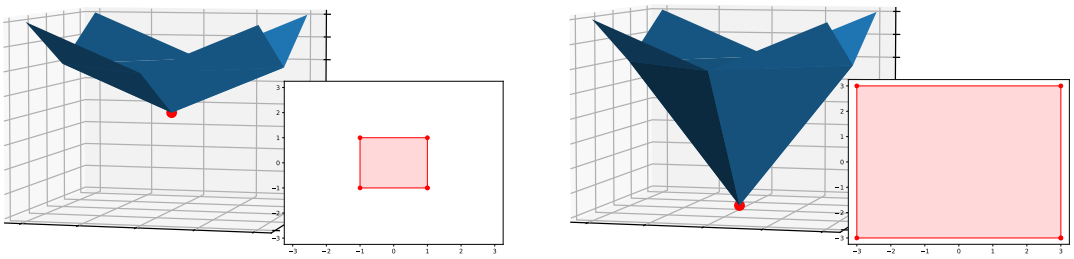
Das Oliker-Prussner-Verfahren ist ein Iterationsverfahren. Es konstruiert eine Funktionenfolge $(u_k)_{k\in\mathbb{N}} \subset W_M$, welche gegen die diskrete Alexandrov-Lösung u_h (nach Definition 2.3.2) konvergiert[1].

Das Verfahren arbeitet dabei sehr geometrisch und nutzt die geometrische Interpretation des Subgradienten, da dieser gewissermaßen die Krümmung bzw. Auswölbung eines Funktionsgraphen beschreibt. Man kann also sagen, dass das Oliker-Prussner-Verfahren eine Funktion anhand eines vorgegebenen Monge-Ampère-Maßes μ, also einer vorgegebenen Intensität der Krümmung bzw. der Auswölbung rekonstruiert.

Wir starten mit einer Funktion $u_0 \in W_M$, welche im Inneren nicht ausgewölbt ist. Damit meinen wir, dass an allen inneren Punkten die Funktion u_0 ein Monge-Ampère-Maß von 0 hat. Alle inneren Punkte sind also entweder Kantenpunkte oder Flächenpunkte, aber keine Eckpunkte des Graphen von u_0 (vgl. Abb. 2.2). Die gesuchte Lösungsfunktion u_h hat an mindestens einem Punkt a_i im Inneren einen Subgradienten größer 0, und ist folglich ausgewölbter als die Startfunktion u_0. Da sowohl u_0 als auch u_h nach unserer Vorbetrachtung konvexe Funktionen sind, müssen die Funktionswerte von u_h

[1] Da wir uns in diesem Kapitel nahezu ausschließlich mit Funktionen aus W_M befassen, benutzen wir hier folgende Notation: u beschreibt die kontinuierliche Lösungsfunktion, u_h die zugehörige diskrete Lösungsfunktion und u_k für $k \in \mathbb{N}$ beschreibt die Funktion des k-ten Iterationsschritts.

im Inneren also niedriger sein als die von u_0. Für alle inneren Gitterpunkte a_i gilt also $0 = \lambda(\partial u_0(a_i)) \leq \lambda(\partial u_h(a_i))$ und $u_0(a_i) \geq u_h(a_i)$.



(a) **(b)**

Abbildung 3.1.: Exemplarische Wirkung des Herunterziehens der Funktionswerte. **(a)** Eine diskrete Funktion mit zugehörigem Subgradienten am roten Punkt. **(b)** Die gleiche Funktion mit heruntergezogenem roten Punkt und verändertem Subgradient.

Ziehen wir nun von der Funktion u_0 einen Punkt $(a_i, u_0(a_i))$ des Graphen in z-Richtung herunter zu einem neuen Punkt (a_i, z_{neu}), so wird der Graph lokal stärker ausgewölbt und das Monge-Ampère-Maß wird größer (vgl. Abb. 3.1). Dabei wollen wir den Punkt so weit herunterziehen, bis das Monge-Ampère-Maß des lokal ausgewölbten Graphen genau dem vorgegebenem $\mu(a_i)$ entspricht. Für die durch das Verfahren konstruierte Funktionenfolge $(u_k)_{k\in\mathbb{N}} \subset W_M$ muss für jeden inneren Gitterpunkt a_i gelten

$$u_0(a_i) \geq u_1(a_i) \geq u_2(a_i) \geq \cdots \geq u_h(a_i). \tag{3.1}$$

Wir ziehen somit in jedem Iterationsschritt die Funktionswerte $u_k(a_i)$ nach unten und wölben den Graphen S_{u_k} lokal aus, was das Monge-Ampère-Maß erhöht. Somit erhalten wir monoton fallende Folgen $(u_k(a_i))_{k\in\mathbb{N}}$ [2] und in dieser Monotonieeigenschaft liegt im wesentlichen auch die Begründung der Konvergenz des Verfahrens. Da in dieser Arbeit allerdings die Implementierung im Vordergrund steht, verweisen wir für die tatsächlichen Konvergenzbeweise auf die originale Veröffentlichung von Oliker und Prussner [1, Kapitel 3].

Das Oliker-Prussner-Verfahren geht nun wie folgt vor: Da es zu kompliziert ist alle z-Werte von u_k an allen Gitterpunkten a_i gleichzeitig durch eine große globale Operation anzupassen[3], sind wir gezwungen in jedem Iterationsschritt jeden z-Wert $u_k(a_i)$ nach-

[2] Es ist wichtig anzumerken, dass der Subgradient der Funktionenfolge zwar steigt, aber nicht zwangsläufig monoton. Dies liegt an der notwendigen globalen Korrektur, siehe dafür weiter unten im Text.

[3] Die Berechnung des Subgradienten $\lambda(\partial u_k(a_i))$ ist abhängig von der Triangulierung von u_k. Da die Triangulierung von u_0 nicht mit der Triangulierung der Zielfunktion übereinstimmen muss (siehe Abschnitt 3.2.2), ist der Erfolg einer solchen globalen Anpassung fragwürdig.

einander einzeln anzupassen. Haben wir nun einen Punkt $(a_i, u_k(a_i))$ zu einem neuen Funktionswert heruntergezogen (a_i, z_{neu}), so ist der Funktionsgraph i.d.R. nicht mehr konvex (für Details verweisen wir auf Abschnitt 3.3.2, anschaulich sieht man dieses Phänomen in Abb. 3.3).

Da aber die Berechnung des Subgradienten bzw. des Monge-Ampère-Maßes und allgemein das gesamte Oliker-Prussner-Verfahren auf der Konvexität der Funktionen beruht, müssen wir die aktualisierte Funktion $\tilde{u_k}$ korrigieren. Um die Konvexität wiederherzustellen, bilden wir konvexe Hülle der aktualisierten Funktion $\tilde{u_k}$ und nehmen den unteren Rand dieser als korrigierten Funktionsgraphen $S_{\tilde{u_k}}$. Diese globale Operation entspricht im wesentlichen der Konstruktion aus Abbildung 2.1b und 2.1c und wird in Abschnitt 3.2.2 genauer erklärt. Das Monge-Ampère-Maß an unserem gerade heruntergezogenem Punkt wird durch diese globale Operation immer abgeschwächt. Somit sorgen wir gleichzeitig dafür, dass das zu erreichende Monge-Ampère-Maß immer leicht unterschätzt wird, damit garantiert wird, dass wir nicht aus versehen einen Punkt des Graphen zu weit herunterziehen.

Da diese globale Korrektur aber sehr aufwendig und teuer ist, wollen wir diese nicht für jeden einzelnen Gitterpunkt durchführen, sondern bündeln dies. Wir berechnen für jeden Gitterpunkt a_i den neuen z-Wert $z_{\text{neu}}(a_i)$, aber anstatt $u_k(a_i)$ damit zu überschreiben, speichern wir die neuen z-Werte in einen temporären Funktion $u_{\text{temp}}(a_i) := z_{\text{neu}}(a_i)$. Somit kann jeder Wert $z_{\text{neu}}(a_i)$ anhand der konvexen Funktion u_k berechnet werden und wir können die globale Korrektur gebündelt auf u_{temp} anwenden.

Damit steht der grobe Ablauf des Oliker-Prussner-Verfahrens in Algorithmus 1. Im Rest dieses Kapitels werden wir anschließend die Implementierung der einzelnen Punkte von diesem Algorithmus genauer erläutern.

Algorithm 1 Oliker-Prussner-Verfahren

Berechne Startfunktion u_0
while True **do**
  if $\|\lambda(\partial u_k(\cdot)) - \mu(\cdot)\|_\infty < \texttt{tol}$ **then**
    beende das Oliker-Prussner-Verfahren
  end if
  for jeden inneren Gitterpunkt a_i **do**
    if Maß bei a_i unzureichend, also $|\lambda(\partial u_k(a_i)) - \mu(a_i)| > \texttt{tol}$ **then**
      Berechne neuen Wert $u_k(a_i) := z_{\text{neu}}$, so dass $\lambda(\partial u_k(a_i)) = \mu(a_i)$
      speichere $u_{\text{temp}}(a_i) = z_{\text{neu}}$
    end if
  end for
  bilde die konvexe Hülle der Menge $\left\{u_{\text{temp}}(a_i) | \forall i = 1, \ldots, M\right\}$
  nehme unteren Rand dieser konvexen Hülle als neuen Funktionsgraphen u_{k+1}
end while

3.2. Vorbereitung für das Verfahren

Bevor wir mit der Implementierung des eigentlichen Verfahren beginnen können, müssen wir uns noch um drei Dinge Gedanken machen:

1. Wir müssen den Funktionenraum W_M implementieren, also eine geeignete Datenstruktur definieren, welche sowohl die Funktionswerte $u_h(a_i)$ speichert, als auch die Triangulierung des Funktionsgraphen S_{u_h}.
2. Wir müssen die für den Funktionenraum W_M motivierende Konstruktion der Funktionsgraphen anhand der konvexen Hülle (vgl. Abb. 2.1) implementieren.
3. Wir müssen eine geeignete Startfunktion u_0 berechnen, sowie eine Berechnung des diskreten Maßes μ aus der rechten-Seite-Funktion f der Monge-Ampère Gleichung festlegen.

3.2.1. Definition der Triangulierung

Prinzipiell besteht die Datenstruktur, welche die Funktionen aus W_M beschreibt aus den 3 Matrizen bzw. Listen `coordinates`, `triangles` und `adjacencies`. [4]

Wir erinnern uns, dass unser Gebiet $\Omega \subset \mathbb{R}^2$ gegeben war durch die Randpunkte $b_1, \ldots b_N$ zusammen mit den inneren Punkte $b_{N+1} := a_1, \ldots, b_{N+M} := a_M$, wobei die

[4] Die tatsächliche Implementierung enthält natürlich noch ein paar weitere Hilfsdaten (z.B. Zählvariablen und Indexmasken), welche für eine korrekte Indizierung bzw. Iterierung notwendig sind.

Randpunkte einen geschlossenen Polygonzug definieren, welcher den Rand von Ω bildet. Die Koordinaten der $N+M$ Punkte werden in der $(N+M) \times 2$-Matrix `coordinates` gespeichert. Die tatsächlichen Funktionswerte $u(b_i)$ werden in einer getrennten Liste (mit gleicher Sortierung) gespeichert, da diese sich während des Verfahrens ändern, wobei die Koordinaten konstant bleiben.

Ferner erinnern wir uns, dass die „konvexe Punktwolke“[5] $(a_i, u_h(a_i))$ über den unteren Rand der konvexen Hülle dieser Punkte einen eindeutigen Funktionsgraphen S_{u_h} definiert. Der Funktionsgraph besteht aus polygonalen Flächen, deren Eckpunkte jeweils mindestens 3 der Punkte $(a_i, u_h(a_i))$ sind. Diese polygonalen Flächen können (nicht notwendigerweise eindeutig) so weit zerlegt werden, dass der gesamte Funktionsgraph durch nicht überlappende Dreiecke gegeben ist, deren Eckpunkte $(a_i, u_h(a_i))$ sind. Der Funktionsgraph S_{u_h} kann somit als Graph über einer Triangulierung $\mathcal{T}$ von Ω betrachtet werden. Diese Triangulierung speichern wir in der $|\mathcal{T}| \times 3$-Matrix `triangles` ab, wobei $|\mathcal{T}|$ die Anzahl der Dreiecke beschreibt und die r-te Reihe dieser Matrix enthält die Indizes i, j, k der Eckpunkte a_i, a_j, a_k des r-ten Dreiecks $T \in \mathcal{T}$.

Die letzten essentiellen Teile der Datenstruktur sind die Nachbarschaftsbeziehungen. Wir werden sehen, dass viele Berechnungen des Oliker-Prussner-Verfahrens Informationen darüber benötigen, welche Dreiecke an einem Gitterpunkt a_i anliegen. Diese Nachbarschaftsbeziehungen werden in der Liste `adjacencies` gespeichert. Dafür enthält der Eintrag `adjacencies[i]` weitere Listen, welche Pointer zu den benachbarten Dreiecken von a_i im bzw. gegen den Uhrzeigersinn speichern.

Die separate Speicherung der Koordinaten, der Elemente und der Nachbarschaftsbeziehungen ist eine gängige Art und Weise eine solche Triangulierung zu definieren. Wir haben uns bei unserer Implementierung von der Half-Edge-Datenstruktur motivieren lassen. Allerdings ist unsere Triangulierung nicht ansatzweise so komplex und umfangreich wie diese.

3.2.2. Konstruktion der Funktionsgraphen und der Startfunktion

Da es sich beim Oliker-Prussner-Verfahren um ein Iterationsverfahren handelt, benötigen wir eine Startfunktion u_0, um mit dem Verfahren zu beginnen. Bei der Konstruktion dieser Startfunktion werden wir einen wichtigen Algorithmus kennenlernen, welcher uns aus einer Punktwolke eine passende Funktion aus W_M generiert.

Wie in Abschnitt 3.1 erwähnt, soll die Startfunktion u_0 im Inneren nicht ausgewölbt sein, also an jedem inneren Punkt a_i eine Monge-Ampère-Maß von 0 besitzen. Gleichzeitig

[5] Also eine Punktwolke, welche einer konvexen Funktion entstammt, bzw. Randbedingungen einer konvexen Funktion modelliert.

muss u_0 als Funktion von W_M die Randwerte erfüllen. Für jeden Randpunkt b_i muss also $u_0(b_i) = g(b_i)$ gelten. Betrachtet man nun von diesen Randwerten den unteren Rand der konvexen Hülle, so erhält man den Funktionsgraphen S_{u_0} der Startfunktion. Nun müssen die polygonalen Flächen von S_{u_0} noch in Dreiecke zerlegt werden, so dass jeder Punkt a_i einer der Eckpunkte dieser Dreiecke ist (dann kann die Triangulation von S_{u_0} gemäß Abschnitt 3.2.1 gespeichert werden).

Man kann sich nun leicht davon überzeugen, dass das Monge-Ampère-Maß von der so konstruierten Funktion u_0 im Inneren von Ω tatsächlich überall 0 ist. Angenommen es gäbe einen inneren Punkt x mit Monge-Ampère-Maß ungleich 0. Dann hätte der Funktionsgraph S_{u_0} am Punkt $(x, u_0(x))$ nach den geometrischen Eigenschaften des Subgradienten eine Auswölbung. Damit ist $(x, u_0(x))$ also ein strikter Eckpunkt von S_{u_0}. Dies ist aber ein Widerspruch dazu, dass der Funktionsgraph S_{u_0} ausschließlich als (untere) konvexe Hülle der Randpunkte $(b_i, g(b_i))$ konstruiert wurde und somit ausschließlich diese Punkte auch strikte Eckpunkte sein können. Eine detailliertere und rigorose Version dieses Beweises findet sich in [1, Lemma 3.3].

Den Konstruktionsalgorithmus kann man also wie folgt zusammenfassen:

Algorithm 2 Konstruktion des Funktionsgraphen anhand der konvexen Hülle

1. berechne die konvexe Hülle der Eingabepunkte $x_1, \ldots, x_k$
2. entferne den oberen Rand aus der konvexen Hülle
3. Zerlege die polygonalen Flächen in Dreiecke
4. stelle sicher, dass jede Stelle a_i auch ein Eckpunkt in dieser Triangulation ist

Wendet man Algorithmus 2 auf die Randpunkte $(b_i, g(b_i))$ an, so erhalten wir die benötigte Startfunktion u_0. Allerdings hat diese Konstruktion eine viel wichtigere Rolle im Oliker-Prussner-Verfahren. Wendet man den Algorithmus 2 nämlich auf die Punkte $(a_i, u_{\text{temp}}(a_i))$ an, so erhalten wir die in Algorithmus 1 genannte globale Korrektur. Dank der Konstruktion über die konvexe Hülle erhalten wir aus Punktwolke $(a_i, u_{\text{temp}}(a_i))$ wieder einen eindeutigen konvexen Funktionsgraphen.

Für die Implementierung gehen wir wie folgt vor: Die Berechnung der konvexen Hülle der Punktwolke wurde mit dem *QuickHull-Algorithmus* implementiert. In Python kann man diesen Algorithmus über die Bibliothek *SciPy* [6] nutzen, wobei diese wiederum intern die C++-Bibliothek *QHull* [7] verwendet. Diese QuickHull-Implementierung liefert uns den Rand der konvexe Hülle automatisch als Dreiecksgebilde (vgl. Abb. 3.2a und 3.2b).

[6] `https://scipy.org`, den QuickHull-Algorithmus findet man unter `scipy.spatial.ConvexHull`
[7] `http://www.qhull.org`

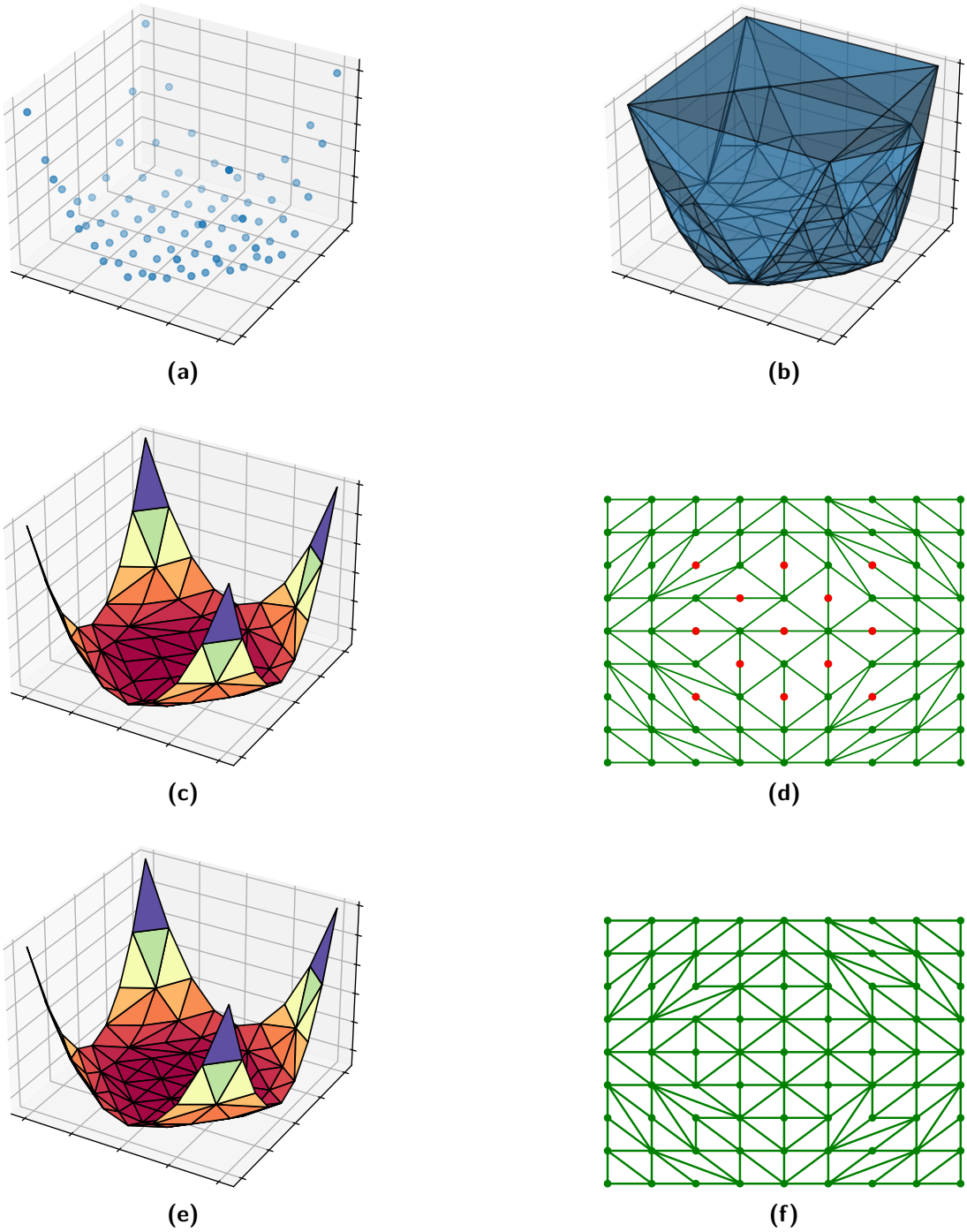


Abbildung 3.2.: Beispielhafte Visualisierung von Algorithmus 2. **(a)** Eine Eingabepunktwolke und **(b)** die daraus mit QuickHull berechnete konvexe Hülle. **(c)** Der untere Rand dieser konvexen Hülle mit **(d)** der daraus resultierenden Triangulierung von Ω. Man beachte, dass die roten Gitterpunkte noch kein echter Bestandteil der Triangulierung sind. **(e)** Der verfeinerte Funktionsgraph, welcher an jedem Gitterpunkt einen Eckpunkt besitzt, mit **(f)** der zugehörigen, verfeinerten Triangulierung von Ω.

Aus dieser konvexen Hülle müssen wir nun den Teil entfernen, der nicht zum unteren Rand gehört und somit nicht für den Funktionsgraphen genutzt wird (vgl. Abb. 3.2b und 3.2c). Die überflüssigen Dreiecke der konvexen Hülle gehören entweder zum seitlichen Rand (Normalenvektor parallel zur x-y-Ebene) oder zum oberen Rand (Dreieck liegt in z-Richtung oberhalb anderer Dreiecke). Die seitlichen Dreiecke können über die Normalenvektoren aussortiert werden. Für die oberen Dreiecke verwenden wir einen Raytracing-Ansatz: Wir betrachten von jedem Gitterpunkt a_i eine Gerade in z-Richtung und schauen mit welchen Dreiecken der konvexen Hülle diese Gerade kollidiert (also Schnittpunkte besitzt). Schneidet der Strahl mehr als ein Dreieck, so muss das (in z-Richtung) obere Dreieck zum oberen Rand der konvexen Hülle gehören und kann entfernt werden.

Zuletzt muss nur noch sichergestellt werden, dass jeder Gitterpunkt a_i tatsächlich auch ein Knoten der Triangulierung ist. Der QuickHull-Algorithmus liefert nämlich als konvexe Hülle eine Triangulierung, welche ausschließlich tatsächlichen Eckpunkte der konvexen Hülle als Knotenpunkte beinhaltet. Da aber nicht jeder Eingabepunkt notwendigerweise ein strikter Eckpunkt der konvexen Hülle ist, müssen diese Punkte der Triangulierung nachträglich wieder hinzugefügt werden. Dies machen wir indem die entsprechenden Dreieck lokal mittels einer Delaunay-Triangulierung verfeinert werden (vgl. Abb. 3.2d und 3.2f bzw. Abb. 3.2c und 3.2e).

3.2.3. Berechnung des diskreten Maßes μ

Die letzte Vorbereitung, die benötigt wird, ist die Berechnung des Ziel-Monge-Ampère-Maßes $\mu(a_i), i = 1, \ldots, M$ aus der Funktion f, welche über die zu lösende Monge-Ampère-Gleichung gegeben ist.

In Abschnitt 2.3 haben wir $\mu(a_i)$ als Integral von f über Partitionsmengen von Ω definiert (vgl. Gleichung 2.9). Diesen theoretischen Ansatz möchten wir leicht abwandeln, aber im Kern treu bleiben.

Wir nehmen dazu die Gitterpunkte $a_i \in \Omega$ und bilden mit ihnen eine Delaunay-Triangulierung $\hat{\mathcal{T}}$ von Ω [8]. Anstatt für die Integrationsbereiche disjunkte Partitionsmengen zu nutzen, verwenden wir die benachbarten Dreiecke der Punkte a_i. Sei also $\hat{\mathcal{T}}_i \subset \hat{\mathcal{T}}$ die Menge der Dreiecke, welche a_i als Eckpunkt haben (Nachbardreiecke von a_i) und für jedes Dreieck $T_{i,j} \in \hat{\mathcal{T}}_i$ sei $\varphi_{i,j} : T_{i,j} \to \mathbb{R}_0^+$ eine lineare Funktion mit $\varphi_{i,j}(a_i) = 1$, welche auf dem Dreieck $T_{i,j}$ linear fällt und in den anderen beiden Eckpunkten den Wert 0 annimmt. Damit kann die Integration für μ_i über f mit den Hutfunktionen $\varphi_{i,\cdot}$ gewichtet

[8] Diese Triangulierung $\hat{\mathcal{T}}$ in diesem Abschnitt ist nicht zu verwechseln mit der Triangulierung $\mathcal{T}$ des Funktionsgraphen welche in allen andern Kapiteln verwendet wird.

werden und wir erhalten

$$\mu_i := \mu(a_i) := \sum_{j=1}^{|\hat{\mathcal{T}}|} \int_{T_{i,j}} \varphi_{i,j} f \, \mathrm{d}x. \tag{3.2}$$

Für die näherungsweise Berechnung dieses Integrals nutzten wir eine auf dem Dreieck symmetrische Quadraturmethode zweiter Ordnung aus [11, Seite 415]. Seien dafür $a_{i,j,1}$, $a_{i,j,2}$ die beiden anderen Eckpunkte des Dreiecks $T_{i,j}$, dann nutzen wir für die Quadratur Auswertungen von $\varphi_{i,j} f$ an den Mittelpunkten der drei Dreiecksseiten:

$$\int_{T_{i,j}} \varphi_{i,j} f \, \mathrm{d}x \approx \frac{\lambda(T_{i,j})}{3} \left((\varphi_{i,j} f) \left(\frac{a_i + a_{i,j,1}}{2} \right) + (\varphi_{i,j} f) \left(\frac{a_i + a_{i,j,2}}{2} \right) + (\varphi_{i,j} f) \left(\frac{a_{i,j,1} + a_{i,j,2}}{2} \right) \right). \tag{3.3}$$

Per Definition ist $\varphi_{i,j}$ auf der Kante von $a_{i,j,1}$, $a_{i,j,2}$ gleich Nulll und in den Mittelpunkten der beiden anderen Kanten gleich $1/2$. Damit vereinfacht sich die Quadratur zu

$$\int_{T_{i,j}} \varphi_{i,j} f \, \mathrm{d}x \approx \frac{\lambda(T_{i,j})}{6} \left(f \left(\frac{a_i + a_{i,j,1}}{2} \right) + f \left(\frac{a_i + a_{i,j,2}}{2} \right) \right). \tag{3.4}$$

Wobei wir das Lebesguemaß $\lambda(T_{i,j})$, also den Flächeninhalt des Dreiecks $T_{i,j}$, über die klassische Determinantenformel für Dreiecke berechnen:

$$\lambda(T_{i,j}) = \frac{1}{2} \left| \det(a_{i,j,1} - a_i, a_{i,j,2} - a_i) \right|. \tag{3.5}$$

3.3. Implementierung des Iterationsschritts

Nachdem wir eine Startfunktion $u_0 \in W_M$ mit zugehöriger Triangulation des Funktionsgraphen konstruiert und ein Zielmaß μ berechnet haben, können wir nun mit dem eigentlichen Iterationsverfahren beginnen. Dabei berechnen wir iterativ aus u_0 eine Funktion deren Monge-Ampère-Maß an den Stellen a_i genau den Wert $\mu(a_i)$ hat[9] und wollen nun insbesondere die For-Schleife des Algorithmus 1 näher beschreiben.

[9] oder aufgrund der globalen Korrektur leicht unterschätzt wird

3.3.1. Berechnung des Monge-Ampère-Maßes einer diskreten Funktion

Sowohl für die Abbruchbedingung (Zeile 3 in Algorithmus 1), als auch für die Prüfung, ob ein Gitterpunkt angepasst werden muss (Zeile 7 in Algorithmus 1), muss das Monge-Ampère-Maß für eine diskrete Funktion $u_m \in W_M$ berechnet werden. [10] Nach Konstruktion des Raumes W_M nach Definition 2.3.1 ist das Monge-Ampère-Maß $\lambda(\partial u_m(\cdot))$ höchstens an den Stellen a_i ungleich Null, sodass wir uns bei der Berechnung auf genau diese Stellen konzentrieren müssen.

Nach Definition 2.2.1 ist der Subgradient $\partial u_m(a_i)$ die Menge aller Steigungen (p, q) von Stützebenen, welche am Graphen S_{u_m} am Punkt $(a_i, u_m(a_i))$ anliegen. Da der Graph S_{u_m} für $u_m \in W_M$ aus Dreiecksflächen gegeben ist, ist die Steigung einer Stützebene an der Stelle a_i durch die Steigungen der Nachbardreiecke von a_i beschränkt. Gleichzeitig beschreiben die Ebenen der Nachbardreiecke genau die Grenzfälle von Stützebenen. Somit ist die Menge $\partial u_m(a_i)$ ein konvexes Polygon, dessen Eckpunkte genau die Steigungen (p, q) der Nachbardreiecke von a_i sind (vgl. Abbildung 2.2, Punkt 3).

Für die konkrete Berechnung von $\partial u_m(a_i)$ sei wieder $\mathcal{T}_i$ die Menge der Nachbardreiecke von a_i, wobei die einzelnen Dreiecke $T_{i,j}$ gegen den Uhrzeigersinn sortiert sind. Ferner sind p_{ij}, q_{ij} die Steigungen in x- bzw. y-Richtung des Dreiecks $T_{i,j}$ im Graphen S_{u_m}. Um nun den Flächeninhalt der Menge $\partial u_m(a_i)$ zu berechnen (also das Monge-Ampère-Maß), betrachten wir also die Dreiecke bestehend aus den Punkten $(0,0)$, (p_{ij}, q_{ij}), (p_{ij+1}, q_{ij+1}) und summieren über die vorzeichenbehafteten [11] Flächeninhalte dieser Dreiecke, welche wir über die Determinantenformel berechnen:

$$\lambda(\partial u_m(a_i)) = \sum_{j=1}^{|\mathcal{T}_i|} \frac{1}{2} \det \begin{pmatrix} p_{ij} & p_{ij+1} \\ q_{ij} & q_{ij+1} \end{pmatrix} = \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i|} (p_{ij} q_{ij+1} - q_{ij} p_{ij+1}) \,. \tag{3.6}$$

Betrachten wir also die Berechnung der Steigungen (p_{ij}, q_{ij}) des Dreiecks T_j. Sei dafür E eine allgemeine Ebene, gegeben durch den Punkt (x_p, y_p, z_p) sowie die Steigungen p, q in x- bzw. y-Richtung. Dann gilt für jeden anderen Punkt $(x, y, z) \in E$ die elementargeometrische Ebenengleichung

$$z = z_p + (x - x_p)p + (y - y_p)q,$$

[10] Dieser Unterabschnitt beschreibt im wesentlichen die Berechnung aus [1, Abschnitt 2.4] angepasst an unsere Triangulierung/Implementierung

[11] Beachte, dass der Punkt $(0,0)$ nicht notwendigerweise im Inneren der Menge $\partial u_m(a_i)$ liegt und demnach einige dieser Teilflächeninhalte positives und andere negatives Vorzeichen haben. Summiert ergibt sich dann genau der Flächeninhalt der Menge $\partial u_m(a_i)$. Dafür müssen die Dreiecke zwingend gegen den Uhrzeigersinn sortiert sein!

welche sich umstellen lässt zu

$$0 = \begin{pmatrix} x - x_p \\ y - y_p \\ z - z_p \end{pmatrix} \cdot \begin{pmatrix} p \\ q \\ -1 \end{pmatrix}.$$

Da der Vektor $(x - x_p, y - y_p, z - z_p)^\top$ ein Richtungsvektor innerhalb der Ebene E ist, sind die Steigungen p, q also Teil des Normalenvektors der Ebene.

Um die Steigung (p_{ij}, q_{ij}) des Dreiecks $T_{i,j}$ zu berechnen, müssen wir also den Normalenvektor mit normierter z-Koordinate des Dreiecks berechnen. Sind also $A_i := (a_i, u(a_i)), A_{ij} := (a_{ij}, u(a_{ij})), A_{ij} := (a_{ij+1}, u(a_{ij+1}))$ die Eckpunkte des Dreiecks $T_{i,j} \subset \mathbb{R}^3$, so erhalten wir den Normalenvektor über das Kreuzprodukt

$$\overline{N_{i,j}} := (A_{ij} - A_i) \times (A_{ij+1} - A_i). \tag{3.7}$$

Nun muss der Normalenvektor nur noch in der z-Koordinate normiert werden[12]:

$$(p_{ij}, q_{ij}, -1) =: N_{i,j} = -\frac{\overline{N_{i,j}}}{\overline{N_{i,j}}^{(2)}}. \tag{3.8}$$

3.3.2. Anpassung der Funktionswerte

Wir widmen uns nun dem Herzstück des Oliker-Prussner-Verfahrens: Dem zentralen Teilalgorithmus, welcher die neuen z-Werte berechnet und somit Zeile 8 aus Algorithmus 1 umsetzt.

Dafür betrachten wir die Funktion u_m nach m Iterationsschritten und die zugehörige Triangulierung $\mathcal{T}$ des Graphen S_{u_m}. Wie in Algorithmus 1 beschrieben, berechnen wir die neuen z-Werte für jeden Gitterpunkt separat, sodass wir uns hier auf einen festen Gitterpunkt a_i, mit $\lambda(\partial u_m(a_i)) < \mu(a_i)$, beziehen. Für diesen Gitterpunkt wollen wir den Funktionswert $z_{\text{alt}} := u_m(a_i)$ nun so verändern, dass das Monge-Ampère-Maß an diesem Punkt des Graphen den Wert $\mu(a_i)$ annimmt.

Dafür führen wir die parametrisierte Funktion u_z mit Triangulierung $\mathcal{T}(z)$ des Graphen S_{u_z} ein, welche prinzipiell gleich zu u_m ist, nur ersetzen wir den Funktionswert $u_m(a_i)$ mit dem Parameter z. Wie in Abschnitt 3.1 erwähnt, wollen wir den Punkt (a_i, z) nun

[12] In unseres Implementierung verzichten wir auf das Minus auf der rechten Seite von Gleichung 3.8, da es aufgrund des symmetrischen Aufbaus von Gleichung 3.6 zur Berechnung des Subgradienten egal ist, ob die z-Koordinate auf 1 oder -1 normiert wird. Wichtig ist nur, dass immer gleich normiert wird!

in z-Richtung herunterziehen, sodass wir generell nur an neuen z-Werten im Intervall $(-\infty, z_{alt}]$ interessiert sind. Somit wird sichergestellt, dass wir mit diesem Verfahren eine monoton fallende Folge $(u_m(a_i))_{m\in\mathbb{N}}$ generieren.

Für die Berechnung des Monge-Ampère-Maßes von u_z am Punkt a_i benötigen wir die Nachbarschaft $\mathcal{T}_i(z) \subset \mathcal{T}(z)$. Für die nachfolgenden Berechnungen ist es wichtig, dass diese Nachbarschaft des Graphen konvex ist. Zwar starten wir mit einem konvexen Gebilde (für $z = z_{\text{alt}}$ ist $u_z = u_m$ konvex), aber wenn wir den Punkt $(a_i, u_z(a_i) = z)$ herunterziehen, dann kann es passieren, dass ab einem Bestimmten z-Wert die lokale Nachbarschaft $\mathcal{T}_i(z)$ konkav wird! Das ist dann der Fall, wenn zwei benachbarte Dreiecke aus $\mathcal{T}_i(z)$ sich beim herunterziehen von z in der Steigung so weit nähern, dass sie irgendwann in einer Ebene liegen und danach zu einer Konkavität in der Nachbarschaft führen (vgl. Abb. 3.3).

Dies ist ein Anzeichen dafür, dass die Kante, an der sich die beiden Dreiecke treffen, in der fertigen Lösungsfunktion nicht existiert und für die Berechnung des lokalen Subgradienten entfernt werden sollte. Da wir nur endlich viele Nachbarpunkte in der Nachbarschaft $\mathcal{T}_i(z)$ haben, können wir also maximal endlich viele Punkte entfernen und werden zwangsweise zu einer geeigneten Nachbarschaft kommen. Es gibt also k viele Werte α mit

$$z_{\text{alt}} =: \alpha_0 > \alpha_1 > \cdots > \alpha_k > \alpha_{k+1} := -\infty, \tag{3.9}$$

an denen zwei Dreiecke der aktuellen Nachbarschaft in eine Ebene fallen und bei weiterem herunterziehen des z-Wertes zu einer Konkavität führen würden. Innerhalb der Intervalle (α_n, α_{n+1}) ist die Nachbarschaft $\mathcal{T}_i(z)$ des Punktes (a_i, z) konstant. Das bedeutet, dass wir dort das Monge-Ampère-Maß $\lambda(\partial u_z(a_i))$ in Abhängigkeit von z berechnen können und die Gleichung $\lambda(\partial u_z(a_i)) = \mu(a_i)$ nach z auflösen können. Liegt die Lösung dieser Gleichung in dem Intervall (α_n, α_{n+1}), mit welchem die Gleichung aufgebaut wurde, so haben wir den gesuchten z-Wert gefunden. Befindet sich die Lösung allerdings in einem anderen Intervall (insbesondere also $\alpha_{n+1} > z_{\text{solution}}$) dann müssen wir die entsprechende Kante, an der die Dreiecke bei $z = \alpha_{n+1}$ zu einer Ebene zusammenfallen, aus der Nachbarschaft entfernen und mit der Berechnung von vorne beginnen. Wir erhalten also für den Gitterpunkt a_i den Algorithmus 3.

Für die tatsächliche Implementierung ist es ausreichend, wenn die Nachbarschaft $\mathcal{T}_i(z)$ als Liste der (gegen den Uhrzeigersinn sortierten) Nachbarpunkte vorliegt, da die Nachbarschaftsinformation in der Reihenfolge dieser Liste liegt und sämtliche weiteren Informationen aus den Koordinaten der Punkte berechnet werden können.

Das Herzstück ist dabei die Berechnung des α-Wertes in Zeile 4 und das Aufstellen der Maßgleichung in Zeile 5 von Algorithmus 3. Für die Maßgleichung verfahren wir analog zu Abschnitt 3.3.1 wobei wir allerdings die Steigungen der Nachbardreiecke in Abhängigkeit der unbekannten z-Koordinate berechnen. Analog zu Gleichung 3.6 erhalten wir dann

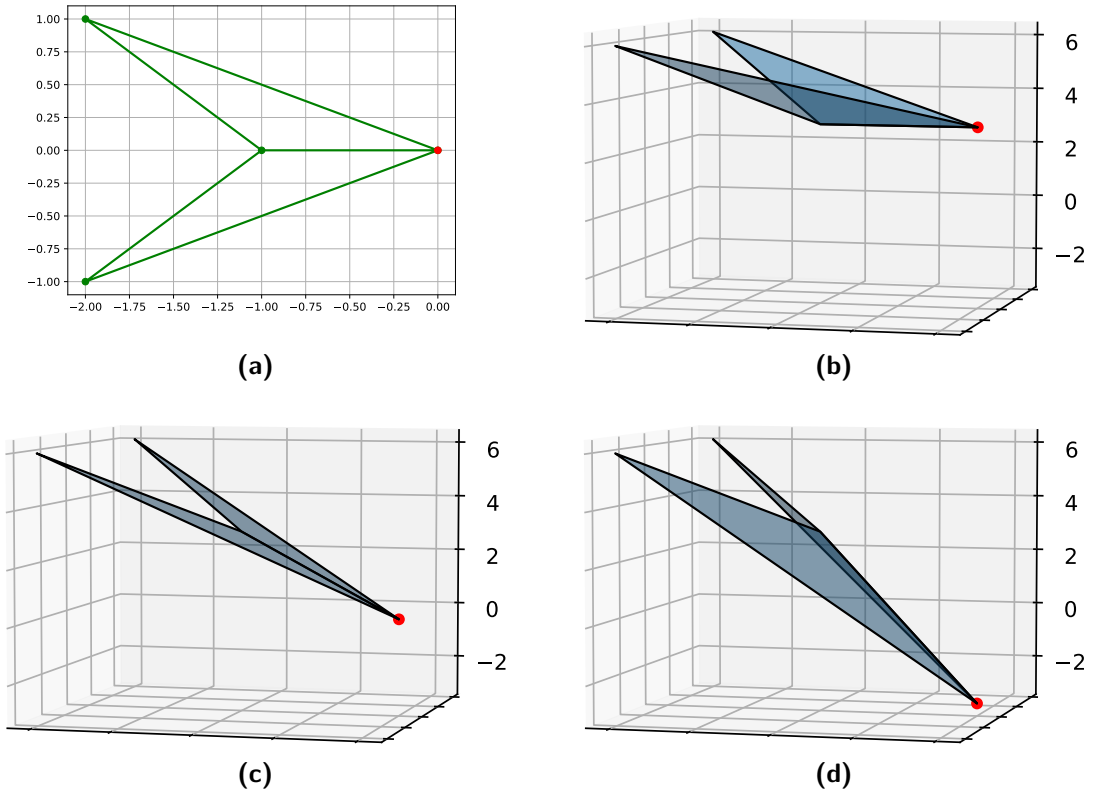


Abbildung 3.3.: Beispielhafte Entstehung der lokalen Konkavität. **(a)** Ein Ausschnitt der Triangulation $\mathcal{T}(z_{alt})$ von u_m. u_z wir im Verlauf am roten Punkt heruntergezogen und dort wird sich in S_{u_z} eine Konkavität bilden. **(b)** Der konvexe Funktionsgraph von u_z vor dem Herunterziehen. **(c)** Nach einigem Herunterziehen des roten Punktes wurde der Funktionsgraph von u_z planar. **(d)** Nach noch weiterem Herunterziehen des roten Punktes wurde der Funktionsgraph von u_z konkav.

das Monge-Ampère-Maß am Punkt a_i in Abhängigkeit der z-Koordinate von u_z:

$$\lambda(\partial u(a_i)) = \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} (p_{ij}(z) q_{ij+1}(z) - q_{ij}(z) p_{ij+1}(z)) \,. \tag{3.10}$$

Da die Steigungen $p(z), q(z)$ nur linear von z abhängen, ist Gleichung 3.10 also quadratisch in z (für eine genauere Beschreibung der Gleichung sowie Lösbarkeitsbeweis verweisen wir auf [1, Abschnitt 3.4.2]). Um diese Gleichung in unserer Implementierung lösen zu können, müssen wir die einzelnen Koeffizienten der quadratischen Gleichung aus den Koordinaten der Nachbarschaftspunkte aus $\mathcal{T}_i(z)$ berechnen und die Gleichung

Algorithm 3 Anpassung des Funktionswertes von a_i

1: Lade die Nachbarschaft $\mathcal{T}_i(z)$ von a_i
2: setze $\alpha_{\text{start}} := z_{\text{alt}}$
3: **while** keine Lösung im aktuellen Intervall gefunden **do**
4: Berechne α_{end} als denjenigen z-Wert, für den zuerst zwei benachbarte Dreiecke von $\mathcal{T}_i(z)$ zu einer Ebene zusammenfallen
5: Baue die Gleichung $\lambda(\partial u_z(a_i)) = \mu(a_i)$ zusammen und löse sie nach z
6: **if** Lösung ist im Intervall $(\alpha_{\text{start}}, \alpha_{\text{end}})$ **then return** Lösung
7: **else**
8: Entferne die zu α_{end} zugehörige Kante (und den Eckpunkt) aus der Nachbarschaft $\mathcal{T}_i(z)$
9: setze $\alpha_{\text{start}} := \alpha_{\text{end}}$
10: Wiederhole den Prozess
11: **end if**
12: **end while**

anschließend per pq-Formel oder anderen Nullstellenalgorithmen lösen. Da die dabei resultierenden Formeln für die Koeffizienten der quadratischen Gleichung sehr unschöne Terme ergeben, welche den Lesefluss hier erheblich stören würden, verweisen wir dafür auf Anhang B.

Für die Berechnung des α-Wertes, müssen wir feststellen, wann zwei benachbarte Dreiecke in einer Ebene zusammenfallen. Zwei aufeinanderfolgende Nachbardreiecke von a_i liegen insbesondere in einer Ebene, wenn ihre Normalenvektoren linear abhängig sind. Wir müssen also folgende Gleichung nach α lösen:

$$\begin{pmatrix} p_{i,j}(\alpha) \\ q_{i,j}(\alpha) \\ -1 \end{pmatrix} = \begin{pmatrix} p_{i,j+1}(\alpha) \\ q_{i,j+1}(\alpha) \\ -1 \end{pmatrix}. \tag{3.11}$$

Somit erhalten wir für je zwei aufeinanderfolgende Nachbardreiecke von a_i maximal eine Lösung α. Der größte all dieser Werte ist dann der gesuchte Wert für α_{end}. Auch hier bauen wir uns die Koeffizienten der Gleichung zusammen und implementieren anschließend einen simplen Löser für diese lineare Gleichung. Da aber auch hier die entstehenden Gleichung den Lesefluss erheblich stören würden, verweisen wir ebenfalls auf den Anhang B.

3.4. Performance und Parallelisierbarkeit

Sei $k := N + M$ die Anzahl der Gitterpunkte und sei $|\mathcal{T}| \in \mathcal{O}(k)$[13] die Anzahl der Dreiecke in der Triangulierung unserer k Gitterpunkte.

Die Laufzeitabschätzung des Oliker-Prussner-Verfahrens ist i.A. sehr schwierig und nur grob möglich, da sich viele Faktoren im Laufe des Verfahrens ändern und man nicht immer pauschal sagen kann, wie viele Schleifendurchläufe bei einer gewissen Operation im Verlauf des Verfahrens erwartbar sind. Für Algorithmus 3, also die lokale Berechnung der neuen Funktionswerte, kamen Oliker und Prussner auf eine durchschnittliche Laufzeit von $\mathcal{O}(\mathrm{NA}(a_i))$, wobei $\mathrm{NA}(a_i)$ die Anzahl der Nachbarpunkte vom Gitterpunkt a_i ist [1, Appendix A]. Da man in vielen Anwendungen und Beispielen von einer ausreichend regulären Triangulation ausgehen kann, kann man für diese Fälle die lokale Berechnung also mit $\mathcal{O}(1)$ pro Gitterpunkt abschätzen. Bei k Gitterpunkten ist dieser Teil eines Iterationsschrittes also grob mit $\mathcal{O}(k)$ (ohne Parallelisierung) erledigt.

Allerdings ist die globale Korrektur der wahre Flaschenhals des Verfahrens(vgl. Zeile 12 und 13 in Algorithmus 1 sowie Abschnitt 3.2.2). Der QuickHull-Algorithmus alleine hat eine Laufzeit von $\mathcal{O}(k^2)$ [13] und dominiert damit bereits die Laufzeit eines Iterationsschritts. Allerdings muss die konvexe Hülle noch angepasst werden. Zum einen muss der nicht-untere Rand entfernt werden und die entstehende Triangulierung muss ggf. verfeinert werden (vgl. Abschnitt 3.2.2). Pro Gitterpunkt, der nachträglich bei der Verfeinerung hinzugefügt werden muss, muss eine $\mathcal{O}(|\mathcal{T}|^2) = \mathcal{O}(k^2)$ Suche angestellt werden, um herauszufinden, in welche Dreiecke der Punkt eingefügt werden muss. Die Anzahl dieser Punkte, welche diese Suche benötigen variiert stark im Verlauf des Verfahrens. Zu Beginn sind es ca. $\mathcal{O}(k)$ und gegen Ende hin $\mathcal{O}(1)$ viele Punkte.

Insgesamt setzt sich jeder Iterationsschritt aus einer $\mathcal{O}(k)$-Berechnung der lokalen Funktionswerte, einer $\mathcal{O}(k^2)$-Berechnung der konvexen Hülle und einer $\mathcal{O}(k^3)$- bzw. $\mathcal{O}(k^2)$-Verfeinerung der Triangulation zusammen. Zusätzlich dazu werden wir im kommenden Kapitel sehen, dass die Anzahl der benötigten Iterationen ebenfalls sehr stark mit k-skaliert. Wie sehr konnten wir allerdings nicht genau quantifizieren.

Oliker und Prussner schlugen deshalb eine zusätzliche Optimierung ihres Verfahrens vor [1, Abschnitt 4]. Sobald die berechnete Lösungsfunktion „nah genug“ an der tatsächlichen Lösung dran liegt, schlugen sie eine Art Newton-Iteration vor, welche dann zu einer schnelleren Konvergenz führt. Da dies aber im Prinzip nur ein Newton-Verfahren ist, war dies kein Bestandteil unserer Untersuchungen.

[13] In einer 2 dimensionalen Delaunay Triangulierung von k Gitterpunkten gibt es maximal $\mathcal{O}(k)$ viele Dreiecke [12]. Da unsere Implementierung zumindest in Teilen auf der Delaunay-Triangulierung basiert, nutzen wir dies als Abschätzung für unser Verfahren.

Des Weiteren haben wir beobachtet, dass es bei bestimmten Funktionen und sehr feinen Gittern zu Instabilitäten im Verfahren kam. Aufgrund der geometrischen Funktionsweise kamen teilweise die QHull-Bibliothek oder unsere Raytracing-Implementierungen an ihre numerischen Grenzen. Da wir dieses Phänomen aber nur vereinzelt beobachten konnten, waren wir aufgrund der langen Rechenzeiten nicht in der Lage dies näher zu quantifizieren.

Da wir für die Erstellung dieser Arbeit nur eine begrenzte Zeit zur Verfügung haben, möchten wir noch kurz zwei mögliche Ansätze zur Optimierung bzw. für weitere Untersuchungen erläutern.

Zum einen könnte man untersuchen, inwiefern es möglich ist das Verfahren erst grob auf einem Teil der Gitterpunkte auszuführen. Somit übernimmt die globale Operation, also der QuickHull-Algorithmus, eine wesentliche Vorarbeit im Verlauf des Verfahrens. Nach einigen Iterationen wendet man dann das Verfahren auf alle Gitterpunkte an, um überall die durch das Monge-Ampère-Maß beschriebene Auswölbung zu erzielen.

Zum anderen könnte man über eine parallelisierte, bzw. Hardware-optimierte Implementierung nachdenken. Zwar gliedert sich jeder Iterationsschritt in zwei Abschnitte, die in einer parallelisierten Version aufeinander warten müssen, aber jeder einzelne ist in sich parallelisierbar. Die lokale Berechnung zu beginn ist per-Design unabhängig von anderen Stützstellen. Der Quick-Hull-Algorithmus funktioniert nach dem Teile-und-herrsche-Ansatz und ist demnach ebenfalls parallelisierbar. Auch die Suche für die Verfeinerung ist unabhängig von der Suche anderer Stützstellen. Ferner kann man auch die Hardwareoptimierung von Grafikkarten für einzelne Teilberechnungen nutzen, wie z.B. dem Raytracing.

4. Numerische Experimente

Als letzten Teil dieser Arbeit wollen wir unsere Implementierung des Oliker-Prussner-Verfahrens testen und die Praktikabilität beurteilen.

Dazu werden wir unsere Software mit den Ergebnissen von Oliker und Prussner aus dem Jahr 1989 vergleichen und anschließend weitere numerische Beispiele besprechen.

Wir werden die Genauigkeit unserer berechneten Funktionen $u_h \in W_M$ immer in der diskreten Unendlichnorm messen $\|u_h - u\|_\infty := \max_{a_i} \{u_h(a_i) - u(a_i)\}$, wobei $u \in W_M$ die exakte diskrete Lösung ist.

Als Abbruchbedingung verwenden wir eine Genauigkeit von 1×10^{-6} im erreichten Monge-Ampère-Maß (vergleiche dazu Zeile 3 in Algorithmus 1).

4.1. Reproduktion der Ergebnisse von Oliker und Prussner

Bei ihrer originalen Veröffentlichung [1] zeigten Oliker und Prussner, dass ihre Implementierung ihres Algorithmus für die zwei Beispielfunktionen

$$u_1(x, y) := x^2 + y^2 \quad \text{und} \tag{4.1}$$

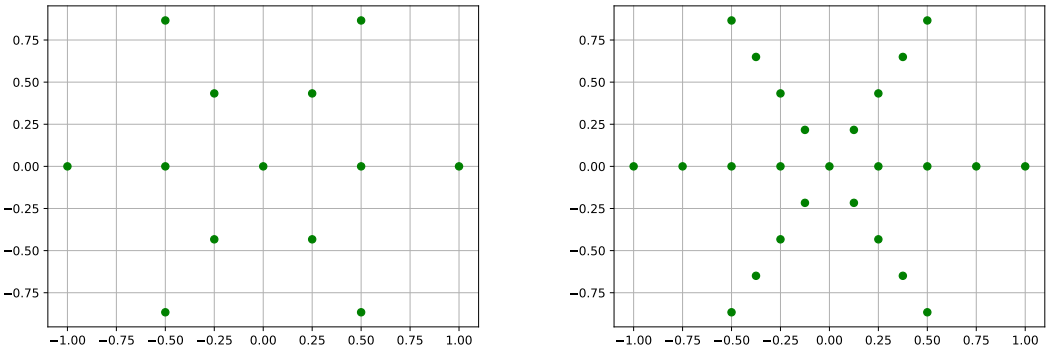
$$u_2(x, y) := \tan\left(x^2 + y^2\right) \tag{4.2}$$

auf den vorgegebenen Gittern von Abb. 4.1 konvergiert und gaben dabei den Fehler in der $\|\cdot\|_\infty$ der berechneten Lösungsfunktion an.

Für die beiden Lösungsfunktionen u_1 und u_2 ergeben sich in der klassischen Monge-Ampère-Gleichung nach 2.1 folgenden Funktionen für die rechte Seite

$$f_1(x, y) = 4 \quad \text{und} \tag{4.3}$$

$$f_2(x, y) = \frac{16(x^2 + y^2)\tan(x^2 + y^2) + 4}{\cos^4(x^2 + y^2)}. \tag{4.4}$$



(a) **(b)**

Abbildung 4.1.: Die Gitterpunkte, welche Oliker und Prussner 1989 verwendeten. Dabei handelt es sich um Diskretisierungen des Einheitskreises **(a)** mit insgesamt 13 Gitterpunkten, je 6 gleichmäßig verteilt auf Kreisen mit Radien 1 und 0.5, sowie einem zusätzlichen Gitterpunkt im Ursprung. Und **(b)** mit insgesamt 25 Gitterpunkten analog zu den Gitterpunkten (a), allerdings mit den Radien $1, 0.75, 0.5, 0.25$.

Allerdings verzichteten Oliker und Prussner darauf die Zielmaße μ aus den Funktionen f zu berechnen[1]. Sie nahmen dabei den diskreten Graphen von u_1 und u_2, berechneten damit das Monge-Ampère-Maß an den Gitterpunkten und nahmen dieses exakte Maß als Eingabemaß μ für das Verfahren.

Der Vergleich unserer Berechnung dieser Beispiele mit den Daten von Oliker und Prussner ist in Tabelle 4.1 abgebildet. Dabei ist zu erkennen, dass wir bis auf eine kleine numerische Ungenauigkeit sämtliche Werte von Oliker und Prussner reproduzieren können. Unter allen vier berechneten numerischen Beispielen beträgt die maximale relative Abweichung von uns zu Oliker und Prussner 2.57%[2].

Ferner kann man beim Vergleich der Berechnungen mit 13 und mit 25 Gitterpunkten erkennen, dass man bei mehr Gitterpunkten auch mehr Iterationen benötigt, um auf den gleichen absoluten Fehler zu kommen. Dies ist eine Beobachtung, welche wir generell für jegliche Ausführung des Verfahrens machen konnten und welche demnach ein treibender Faktor dafür ist, dass die Berechnung großer numerischer Beispiele mit einer (teilweise extrem) langen Rechenzeit verbunden ist.

[1] Zwar kann man dies dem Paper [1] nicht direkt entnehmen, da sie aber nie angaben wie man μ aus f berechnet, kann man davon ausgehen, dass dies kein Bestandteil ihrer Untersuchungen war.

[2] Die relative Abweichung wurde mit mehr Nachkommastellen berechnet, als die Werte in der Tabelle 4.1 abgebildet sind.

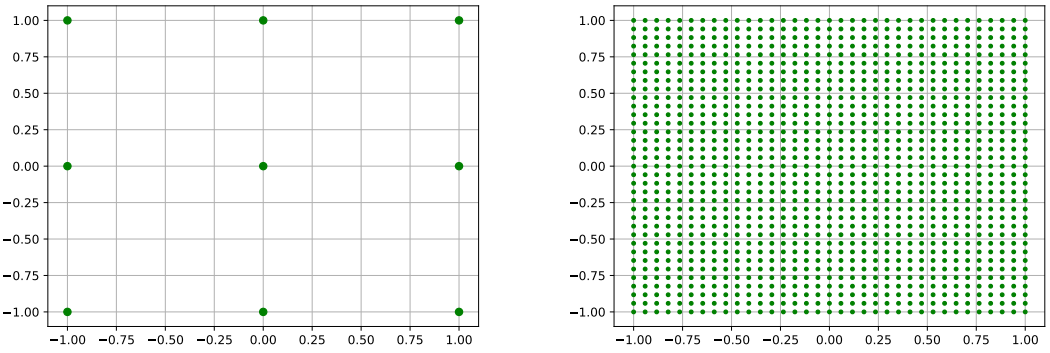
	$m = 5$	$m = 10$	$m = 15$	$m = 20$	$m = 25$
Funktion u_1 mit 13 Gitterpunkten:					
Wir	1.28×10^{-1}	1.86×10^{-2}	2.76×10^{-3}	4.10×10^{-4}	6.10×10^{-5}
O&P	1.10×10^{-1}	1.60×10^{-2}	2.40×10^{-3}	3.50×10^{-4}	5.20×10^{-5}
Funktion u_2 mit 13 Gitterpunkten:					
Wir	1.96×10^{-1}	2.72×10^{-2}	3.85×10^{-3}	5.46×10^{-4}	7.75×10^{-5}
O&P	1.70×10^{-1}	2.30×10^{-2}	3.30×10^{-3}	4.70×10^{-4}	6.80×10^{-5}
	$m = 10$	$m = 20$	$m = 30$	$m = 40$	$m = 50$
Funktion u_1 mit 25 Gitterpunkten:					
Wir	1.92×10^{-1}	5.41×10^{-2}	1.64×10^{-2}	5.04×10^{-3}	1.56×10^{-3}
O&P	1.80×10^{-1}	5.20×10^{-2}	1.60×10^{-2}	4.80×10^{-3}	1.50×10^{-3}
Funktion u_2 mit 25 Gitterpunkten:					
Wir	1.70×10^{-1}	3.44×10^{-2}	7.62×10^{-3}	1.72×10^{-3}	3.89×10^{-4}
O&P	1.60×10^{-1}	3.20×10^{-2}	7.20×10^{-3}	1.60×10^{-3}	3.70×10^{-4}

Tabelle 4.1.: Vergleich der Genauigkeiten unserer Implementierung (Wir) und der von Oliker und Prussner aus [1] (O&P) anhand der Funktionen u_1, u_2 sowie den Gitterpunkten aus Abb. 4.1. Die dargestellten Werte sind die an den Gitterpunkten ausgewerteten diskreten Normen $\|u_m - u_1\|_\infty$ bzw. $\|u_m - u_2\|_\infty$ nach m Iterationsschritten.

4.2. Weitere numerische Experimente

Alle weiteren numerischen Experimente haben wir aufgrund der besseren Skalierbarkeit auf einem quadratischen Gebiet mit äquidistant platzierten Gitterpunkten berechnet (i.d.R. auf $\Omega = [-1, 1]^2$). Um zu untersuchen, wie sich das Verfahren bei steigender Anzahl der Gitterpunkte verhält, haben wir jedes Beispiel auf Gittern mit 3 Gitterpunkten pro Dimension (also insgesamt $3 \cdot 3 = 9$ Punkte) bis hin zu Gittern mit 35 Gitterpunkten pro Dimension (also insgesamt $35 \cdot 35 = 1225$ Punkte) berechnet (Vergleiche Abb. 4.2). Wir haben also Gitterweiten von $h = 0.667$ bis $h = 0.057$ berechnet.

Bei unseren Rechenbeispielen konnten wir im Allgemeinen bemerken, dass die Rechenzeit mit steigender Stützstellenanzahl massiv steigt. Nicht nur weil die globalen Operationen als Flaschenhals des Verfahrens bei mehr Gitterpunkten kubisch skalieren, sondern insbesondere weil das Verfahren deutlich mehr Iterationen benötigt, um die gleiche Approximation des Monge-Ampère-Maßes zu erreichen. So haben wir für 9^2 Gitterpunkte wenige Minuten gebraucht, während wir bei 35^2 Gitterpunkte i.d.R. zwischen 12 und



(a) **(b)**

Abbildung 4.2.: (a) Unser gröbstes verwendetes Gitter mit 3 Gitterpunkten pro Dimension und **(b)** unser feinstes verwendetes Gitter mit 35 Gitterpunkten pro Dimension.

16 Stunden gerechnet haben. Bei höheren Genauigkeiten im Monge-Ampère-Maß, z.B. 10^{-8} anstatt 10^{-6}, haben wir teilweise die 48 Stunden Rechenzeit geknackt.[3]

Beispiel 1: Die bisherigen Beispiele u_1 und u_2 waren glatte Funktionen, welche klassische Lösungen einer Monge-Ampère-Gleichung sind. Deswegen betrachten wir nun die Betragsfunktion

$$u_3(x, y) = 1 - |x + y| \,. \tag{4.5}$$

Diese Funktion ist nicht differenzierbar und somit ist sie auch keine Lösung der Monge-Ampère-Gleichung im klassischen Sinne. Da diese Funktion aber konvex ist und der Subgradient somit überall wohldefiniert ist, können wir im Inneren von $\Omega = [-1, 1]^2$ das Monge-Ampère-Maß berechnen und erhalten folgende Alexandrov-Formulierung der Monge-Ampère-Gleichung

$$\lambda(\partial u_4(x, y)) := 4\delta_{(0,0)}(x, y), \tag{4.6}$$

wobei $\delta_{(0,0)}$ das Diracmaß im Ursprung ist. Das Monge-Ampère-Maß ist also lediglich im Ursprung nicht 0.

Diese Funktion konnte von unserer Implementierung sehr gut berechnet werden. Unabhängig von der Anzahl der Gitterpunkte (hier 5 bis 35 Punkte pro Dimension) haben wir eine Genauigkeit der Unendlichnorm $\|u_h - u_3\|_\infty$ von 1.0×10^{-7} bis 1.2×10^{-7} erreicht

[3] Man beachte, dass Rechenzeiten Systemabhängig sind und wir bereits in Abschnitt 3.4 erwähnt haben, dass bei unserer Implementierung noch viel Spielraum im Performance-Sinne ist. Somit ist dies nur einen Einblick dafür, wie die Rechenzeit grob skalieren kann.

(vgl. Tabelle 4.2). Das besondere an dieser Funktion ist, dass wir bei der Berechnung deutlich weniger Iterationen benötigten als bei allen anderen Funktionen. Zum Vergleich, bei 35 Gitterpunkten pro Dimension haben wir hier nur 262 gebraucht und sonst bis zu 2000 Iterationen!

	$k = 5^2$	$k = 9^2$	$k = 13^2$	$k = 21^2$	$k = 35^2$
Funktion u_3 auf $\Omega = [-1, 1]^2$ mit k Gitterpunkten:					
$\|u_h - u_3\|_\infty$	1.19×10^{-7}	1.01×10^{-7}	1.2×10^{-7}	1.16×10^{-7}	1.19×10^{-7}
#Iterationen	23	56	87	151	262

Tabelle 4.2.: Ergbenisse des Verfahrens für die Funktion u_3 für verschiedene Anzahlen an Gitterpunkten

Beispiel 2: Mit den bisherigen Beispielen ist lediglich demonstriert, dass das Verfahren konvergiert unter der Annahme, dass man das exakte Maß verwendet. Dies funktioniert sowohl für klassische Lösungen, als auch für Funktionen, welche ausschließlich Alexandrov-Lösungen sind. Wir wollen nun zusätzlich demonstrieren, dass unsere Berechnung des Zielmaßes μ aus der Funktion der rechten Seite f aus Abschnitt 3.2.3 funktioniert. Für eine konkrete Wahl der Gitterpunkte werden wir mit dem aus f berechnetem Zielmaß μ nur eine Näherung des exakten Monge-Ampère-Maßes bekommen. Somit werden wir mit steigender Anzahl an Iterationsschritten auch keine Konvergenz zur tatsächlichen Lösungsfunktion sehen. Das Verfahren konvergiert immer noch, nur eben gegen eine falsche Funktion. Folglich werden wir für wenig Gitterpunkte auch einen großen absoluten Fehler sehen.

Die Idee ist nun, dass mit einem feiner werdenden Gitter das aus f berechnete Zielmaß μ gegen das tatsächliche Monge-Ampère-Maß konvergiert und wir dabei eine Abnahme des Fehlers der berechneten Funktion beobachten wollen.

Wir betrachten dafür die Funktionen u_1, sowie eine weitere glatte Funktion

$$u_4(x, y) := \exp\left(x^2 + y^2\right). \tag{4.7}$$

Für die neue Funktion erhalten wir die rechte Seite

$$f_4(x, y) = \exp\left(x^2 + y^2\right) \cdot \left(x^2 + y^2 + 1\right). \tag{4.8}$$

Für diese beiden Funktionen haben wir Rechenbeispiele mit $3, 5, 9, 13, 21$ und 35 Gitterpunkten pro Dimension durchgerechnet. Mittels Abschnitt 3.2.3 können wir die Monge-Ampère-Maße μ_1 und μ_4 aus f_1 bzw. f_4 berechnen. Die Genauigkeit dieser Maße sehen wir in Abb. 4.3. Mit zunehmender Anzahl an (äquidistanten) Gitterpunkten konvergieren

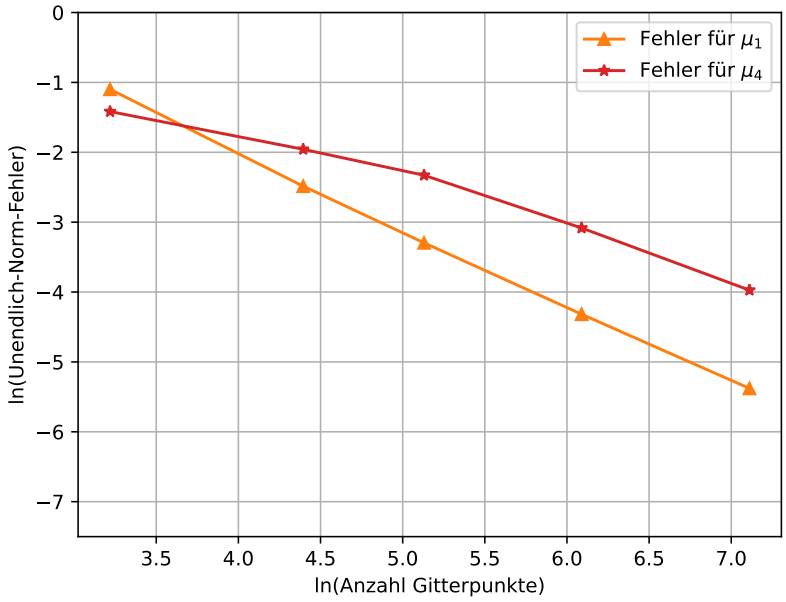


Abbildung 4.3.: Genauigkeit der berechneten Maße μ_1 und μ_4 in Abhängigkeit der Anzahl an Gitterpunkte im Log-Log-Plot.

die Maße μ_1 bzw. μ_4 gegen die Monge-Ampère-Maße der exakten Lösungen u_1 bzw. u_4. Dabei konvergieren μ_1 und μ_4 mit einer Geschwindigkeit von ca. 1 bzw. 0.8 [4].

Die Genauigkeit der berechneten Lösung in der Unendlichnorm sehen wir in Abb. 4.4. Hier können wir den erhofften Effekt sehen, dass mit zunehmender Anzahl an Gitterpunkten (und deshalb genauerem Maß) die berechnete Lösungsfunktion gegen die tatsächliche Zielfunktion u_1 bzw. u_4 konvergiert. Für beide Funktionen erreichen wir eine Konvergenzgeschwindigkeit von ca. 1. Außerdem kann man erkennen, dass die anspruchsvollere Funktion u_4 durchweg etwas schlechter performt.

Besonders bemerkenswert ist aber das Verhalten der berechneten Lösungsfunktion, wenn wir das exakte Monge-Ampère-Maß verwenden. Dort verschlechtert sich die Genauigkeit des Verfahrens mit steigender Anzahl der Gitterpunkte. Dies ist vermutlich darauf zurückzuführen, dass wir das Monge-Ampère-Maß nur mit begrenzter Genauigkeit (hier 10^{-6}) berechnen. Geometrisch gesprochen erklären wir uns das wie folgt: Haben wir mehr Gitterpunkte so haben wir häufiger einen Krümmungsfehler von 10^{-6}. Da das Verfahren das Monge-Ampère-Maß unterschätzt, wird der entstehende Funktionsgraph somit nicht

[4] Mit Konvergenzgeschwindigkeit α bezeichnen wir hier den Zusammenhang zwischen dem Fehler und der Anzahl der Gitterpunkte k: $\ln(\text{Fehler}(k)) \approx \text{const.} - \alpha \ln(k)$

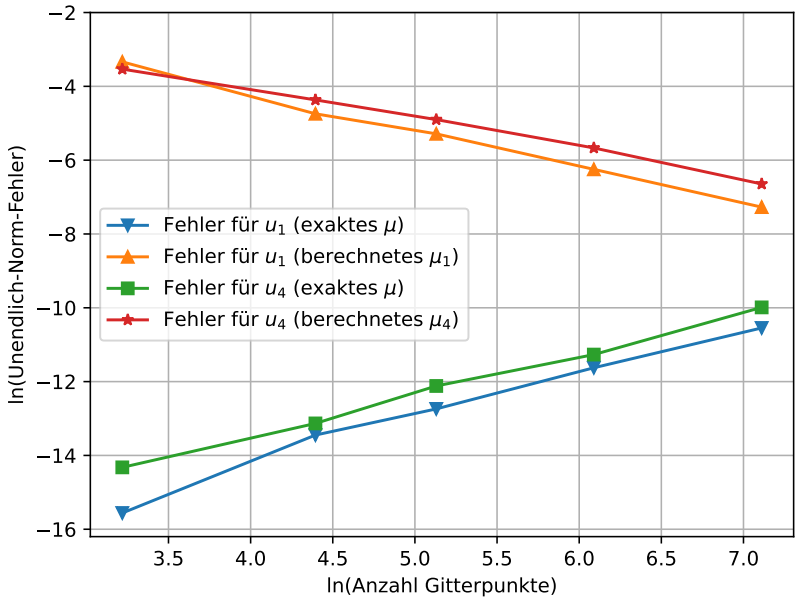


Abbildung 4.4.: Verhalten des Oliker-Prussner-Verfahrens für u_1 und u_4 in Abhängigkeit der Anzahl an Gitterpunkte im Log-Log-Plot. Jeweils unter Nutzung der perfekten Monge-Ampère-Maße und der aus f_1 bzw. f_4 berechneten.

ausreichend ausgewölbt und bei mehr Gitterpunkten tendieren wir dann dazu den Graphen somit noch weniger auszuwölben. Vereinzelte Tests scheinen diese Hypothese zu bestätigen, da man diesem Effekt entgegen steuern kann, wenn man die Genauigkeit erhöht (bspw. auf 10^{-8}).

5. Fazit und Ausblick

Diese Bachelorarbeit hatte zum Ziel, das selten verwendete Oliker-Prussner-Verfahren, als Löser der zweidimensionalen Monge-Ampère-Gleichung, in einer Software umzusetzen, dieses ausgiebig zu testen und gleichzeitig eine detaillierte Anleitung zu schaffen, wie man es algorithmisch implementieren kann. Die dabei entwickelte Software kann die experimentellen Ergebnisse von Oliker und Prussner aus dem Jahr 1989 vollständig reproduzieren und die diskrete Alexandrov-Formulierung aus Definition 2.3.2 für sämtliche Eingaben lösen.

Die größte Herausforderung war, dass Oliker und Prussner das Verfahren in ihrer Veröffentlichung [1] zwar mathematisch exakt formuliert haben, diese eher geometrische Beschreibung aber für eine vollumfängliche Implementierung unzureichend ist. Um diese Ausführung zu erweitern, entwickelten wir unter Nutzung der QuickHull-Bibliothek in Abschnitt 3.2.2 einen eigenen Teilalgorithmus, welcher aus einer Eingabepunktwolke einen konvexen, stückweise linearen Funktionsgraphen erstellt und somit die globale Korrektur des Oliker-Prussner-Verfahrens implementieren kann.

Des Weiteren haben Oliker und Prussner ihr Verfahren lediglich unter der Annahme getestet, dass das korrekte Monge-Ampère-Maß bekannt ist. In einem praktisch relevanten Setting ist dieses Wissen aber a priori nicht vorhanden, sodass man dieses Maß zuerst aus der zu lösenden Monge-Ampère-Gleichung zusammen bauen muss. Auch dafür haben wir einen eigenen Teilalgorithmus entwickelt, welcher dieses Problem mithilfe von mehrdimensionalen numerischen Quadraturmethoden löst.

Obwohl das Verfahren ausreichend glatte Funktionen einwandfrei berechnet und akzeptable Konvergenzraten aufzeigt, so stellten wir bei der anschließenden Untersuchung unserer Software fest, dass ein Fehler in diesem Eingabemaß einen kritischen Einfluss auf das Oliker-Prussner-Verfahren hat.

Auch beim Aspekt der Performance zeigt das Oliker-Prussner-Verfahren, dass es eher zu den rechenintensiven Verfahren zählt. Die Laufzeit eines einzelnen Iterationsschrittes hängt teilweise kubisch von der Anzahl der Gitterpunkte ab. Zusätzlich steigt die Anzahl der benötigten Iterationen stark mit der Anzahl der Gitterpunkte an. Dennoch gestaltet es sich im Allgemeinen schwierig, detaillierte Abschätzungen über die Laufzeit des Verfahrens zu machen. Glücklicherweise gibt es vielversprechende Ansätze, welche dieses Verfahren beschleunigen könnten. Ein Beispiel dafür ist die von Oliker und Pruss-

ner selbst vorgeschlagene Newton-Iteration. Des Weiteren bietet das Verfahren generell viel Spielraum bei der Parallelisierbarkeit. In weiterführenden Arbeiten könnte dies untersucht werden, oder inwiefern man wirklich jeden Gitterpunkt berechnen muss, da man vielleicht durch die globale Operation gerade zum Beginn des Verfahrens viel Rechenaufwand sparen könnte.

Letztendlich kommen wir zu dem Schluss, dass das Oliker-Prussner-Verfahren ein gelungenes und funktionsfähiges Verfahren ist. Dennoch wird es, aufgrund seines benötigten Rechenaufwandes, höchstens auf Großrechnern mit sorgfältig optimiertem und parallelisiertem Quellcode praktikabel sein. Und auch nur unter der Prämisse, dass der Einfluss des fehlerhaft approximierten Monge-Ampère-Maßes auf die berechnete Lösung quantifiziert werden kann.

A. Software

ACHTUNG

Hinweis: Dieses Kapitel des Anhangs wurde in die `README.md`-Datei des Repositorys ausgelagert. Um Inkompatibilitäten zwischen diesem Kapitel und der `README.md`-Datei zu vermieden, wurde dieses Kapitel hier entfernt.

B. Implementierung ausgewählter Formeln

Für das Oliker-Prussner-Verfahren spielen insbesondere die Gleichungen 3.10 und 3.11 eine sehr bedeutsame Rolle. Diese Gleichungen sind zwar aus mathematischer Sichtweise vollkommen korrekt und ausreichend, allerdings aufgrund der Unbekanntheit von z in ihrer aktuellen Formulierung nicht dazu geeignet sie in einem Computerprogramm zu implementieren. Diese technische Herleitung und Verifikation des Quellcodes möchten wir hier der Arbeit anhängen.

B.1. Berechnung der Formel 3.10

Betrachten wir erneut die Gleichung 3.10, welche uns erlaubt das Monge-Ampère-Maß am Gitterpunkt a_i in Abhängigkeit des z-Wertes der Funktion :

$$\lambda(\partial u(a_i)) = \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} \left(p_{ij}(z)q_{ij+1}(z) - q_{ij}(z)p_{ij+1}(z)\right).$$

Wir erinnern uns, dass diese Gleichung eine quadratische Gleichung in z ergibt. Um diese quadratische Gleichung lösen zu können, müssen wir sie also in die Form $\lambda(\partial u(a_i)) = az^2 + bz + c$ bringen und die Koeffizienten a, b, c aus unserem Gitter berechnen[1].

Fixieren wir ein $j \in \{1, \ldots, |\mathcal{T}_i|\}\}$. Dann enthält ein Summand der Summe die Steigungen der zwei benachbarten Dreiecke $T_{i,j}$ und $T_{i,j+1}$ aus der Nachbarschaft $\mathcal{T}_\rangle$ des Graphen am Punkt (x_i, y_i, z). Die Steigungen ergeben sich aus den Normalenvektoren, wenn wir sie in der z-Komponente normieren. Betrachten wir die Berechnung des Normalenvektors $\overline{N_{i,j}}(z)$ des Dreiecks $T_{i,j}$. Das Dreieck hat die Eckpunkte $A_i := (x_i, y_i, z), A_{i,j} := (x_{i,j}, y_{i,j}, z_{i,j}), A_{i,j+1} := (x_{i,j+1}, y_{i,j+1}, z_{i,j+1})$, sodass sich der

[1] Hier sei kurz erwähnt, dass Oliker und Prussner die fertige Formel für die Koeffizienten zwar in [1, Appendix A] erwähnten, sie allerdings nicht begründeten oder näher hergeleitet haben

Normalenvektor aus dem Kreuzprodukt ergibt:

$$\overline{N_{i,j}}(z) = \begin{pmatrix} x_{i,j} - x_i \\ y_{i,j} - y_i \\ z_{i,j} - z \end{pmatrix} \times \begin{pmatrix} x_{i,j+1} - x_i \\ y_{i,j+1} - y_i \\ z_{i,j+1} - z \end{pmatrix} \tag{B.1}$$

$$= \begin{pmatrix} (y_{i,j} - y_i)(z_{i,j+1} - z) - (y_{i,j+1} - y_i)(z_{i,j} - z) \\ (z_{i,j} - z)(x_{i,j+1} - x_i) - (z_{i,j+1} - z)(x_{i,j} - x_i) \\ (x_{i,j} - x_i)(y_{i,j+1} - y_i) - (x_{i,j+1} - x_i)(y_{i,j} - y_i) \end{pmatrix} \tag{B.2}$$

$$= \begin{pmatrix} (y_{i,j} - y_i)z_{i,j+1} - (y_{i,j+1} - y_i)z_{i,j} \\ z_{i,j}(x_{i,j+1} - x_i) - z_{i,j+1}(x_{i,j} - x_i) \\ (x_{i,j} - x_i)(y_{i,j+1} - y_i) - (x_{i,j+1} - x_i)(y_{i,j} - y_i) \end{pmatrix} + z \begin{pmatrix} y_{i,j+1} - y_{i,j} \\ x_{i,j} - x_{i,j+1} \\ 0 \end{pmatrix} \tag{B.3}$$

$$= \overline{N_{i,j}}(0) + zD_z(\overline{N_{i,j}}(z)). \tag{B.4}$$

Hierbei beschreibt $D_z(\overline{N_{i,j}}(z))$ die symbolische, komponentenweise Ableitung des Termes $\overline{N_{i,j}}(z)$. Allerdings werden wir diesen Ausdruck niemals über die Ableitung berechnen, sondern nutzten das hier nur als symbolische Notation. Das Besondere an der Darstellung B.4 ist, dass wir nun den Normalenvektor durch zwei Vektoren darstellen können, welche nicht von z abhängig sind. Für die Implementierung ist diese Darstellung deshalb von Vorteil, da wir für einfach $z = 0$ setzen können, das Kreuzprodukt normal berechnen, und abschließend $D_z(\overline{N_{i,j}}(z))$ als kleinen Korrekturterm definieren.

Um nun die Steigungen $p_{i,j}, q_{i,j}$ zu erhalten, müssen wir $\overline{N_{i,j}}(z)$ in der z-Komponente normieren, also durch $\gamma_j := (x_{i,j} - x_i)(y_{i,j+1} - y_i) - (x_{i,j+1} - x_i)(y_{i,j} - y_i)$ teilen:

$$\begin{pmatrix} p_{i,j}(z) \\ q_{i,j}(z) \\ 1 \end{pmatrix} = N_{i,j} + zD_{i,j} := \frac{\overline{N_{i,j}}(0)}{\gamma_j} + z\frac{D_z(\overline{N_{i,j}}(z))}{\gamma_j} \tag{B.5}$$

Und diese Vorgehensweise findet sich auch so in unserer Implementierung wieder:

```
A_ij = #lade (x_ij, y_ij, z_ij)
A_ij1 = #lade (x_ij+1, y_ij+1, z_ij+1)
A_i = #lade (x_i, y_i, 0)

Nbar_ij = np.cross(A_ij-A_i, A_ij1-A_i)
gamma_ij = Nbar_ij[2]

N_ij = Nbar_ij/gamma_ij
D_ij = np.array([A_ij1[1]-A_ij[1], A_ij[0]-A_ij1[0], 0])/gamma_ij
```

Wiederholen wir das gleiche analog für $N_{i,j+1}$ und $D_{i,j+1}$ des benachbarten Dreiecks $T_{i,j+1}$, so haben wir alle 4 Steigungen aus Formel 3.10 beisammen. Der Übersichtlichkeit halber, setzen wir notationell $N_j^{(k)}$ für die k-te Komponente von $N_{i,j}$, analog $D_j^{(k)}$.

Damit ergibt sich:

$$\lambda(\partial u(a_i)) = \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} (p_{ij}(z)q_{ij+1}(z) - q_{ij}(z)p_{ij+1}(z)) \tag{B.6}$$

$$= \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} \left((N_j^{(1)} + zD_j^{(1)})(N_{j+1}^{(2)} + zD_{j+1}^{(2)}) - (N_j^{(2)} + zD_j^{(2)})(N_{j+1}^{(1)} + zD_{j+1}^{(1)}) \right). \tag{B.7}$$

Multiplizieren wir nun dieses Produkt aus und sortieren wir die Terme nach $1, z, z^2$, so erhalten wir $\lambda(\partial u(a_i)) = az^2 + bz + c$ für folgende Koeffizienten

$$a := \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} \left(D_j^{(1)} D_{j+1}^{(2)} - D_j^{(2)} D_{j+1}^{(1)} \right) \tag{B.8}$$

$$b := \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} \left(D_j^{(1)} N_{j+1}^{(2)} - D_j^{(2)} N_{j+1}^{(1)} + N_j^{(1)} D_{j+1}^{(2)} - N_j^{(2)} D_{j+1}^{(1)} \right) \tag{B.9}$$

$$c := \frac{1}{2} \sum_{j=1}^{|\mathcal{T}_i(z)|} \left(N_j^{(1)} N_{j+1}^{(2)} - N_j^{(2)} N_{j+1}^{(1)} \right) \tag{B.10}$$

Somit können wir problemlos die Koeffizienten mit einer For-Schleife anhand unserer Triangulierung berechnen, anschließend die quadratische Gleichung mittels pq-Formel o.Ä. lösen und somit den gesuchten z-Wert berechnen.

B.2. Berechnung der Formel 3.11

Betrachten wir erneut die Gleichung 3.11, welche berechnet, wann zwei benachbarte Dreiecke in der Nachbarschaft $\mathcal{T}_i(z)$ zu einer Ebene zusammen fallen:

$$\begin{pmatrix} p_{i,j}(\alpha) \\ q_{i,j}(\alpha) \\ -1 \end{pmatrix} = \begin{pmatrix} p_{i,j+1}(\alpha) \\ q_{i,j+1}(\alpha) \\ -1 \end{pmatrix}.$$

Analog zu Abschnitt B.1 können wir zum Berechnen der Steigungen den Normalenvektor $\overline{N_{i,j}}(\alpha)$ bilden und diesen wieder in seine zwei Anteile zerlegen $\overline{N_{i,j}}(\alpha) = \overline{N_{i,j}}(0) - zD_\alpha(\overline{N_{i,j}}(\alpha))$ und auch γ_j wieder definieren. Somit lässt sich die Gleichung schreiben als[2]

$$\begin{pmatrix} p_{i,j}(\alpha) \\ q_{i,j}(\alpha) \\ 1 \end{pmatrix} = \frac{\overline{N_{i,j}}(0)}{\gamma_j} + \alpha \frac{D_\alpha(\overline{N_{i,j}}(\alpha))}{\gamma_j} = \frac{\overline{N_{i,j+1}}(0)}{\gamma_{j+1}} + \alpha \frac{D_\alpha(\overline{N_{i,j+1}}(\alpha))}{\gamma_{j+1}} = \begin{pmatrix} p_{i,j+1}(\alpha) \\ q_{i,j+1}(\alpha) \\ 1 \end{pmatrix}. \tag{B.11}$$

Dabei können wir das α separieren

$$\frac{\overline{N_{i,j}}(0)}{\gamma_j} - \frac{\overline{N_{i,j+1}}(0)}{\gamma_{j+1}} = \alpha \left(\frac{D_\alpha(\overline{N_{i,j+1}}(\alpha))}{\gamma_{j+1}} - \frac{D_\alpha(\overline{N_{i,j}}(\alpha))}{\gamma_j} \right), \tag{B.12}$$

und mit den Nennern durch multiplizieren

$$\overline{N_{i,j}}(0)\gamma_{j+1} - \overline{N_{i,j+1}}(0)\gamma_j = \alpha \left(D_\alpha(\overline{N_{i,j+1}}(\alpha))\gamma_j - D_\alpha(\overline{N_{i,j}}(\alpha))\gamma_{j+1} \right) \tag{B.13}$$

Dann ergibt sich $\alpha = \frac{a}{b}$ für

$$a := \overline{N_{i,j}}(0)\gamma_{j+1} - \overline{N_{i,j+1}}(0)\gamma_j \tag{B.14}$$

$$b := D_\alpha(\overline{N_{i,j+1}}(\alpha))\gamma_j - D_\alpha(\overline{N_{i,j}}(\alpha))\gamma_{j+1}. \tag{B.15}$$

Somit erhalten wir für je zwei benachbarte Dreiecke in unserer Nachbarschaft $\mathcal{T}_i$ einen α-Wert. Es ist aber noch wichtig anzumerken, dass diese Gleichung nicht immer lösbar ist! Denn nicht immer ist es überhaupt möglich, dass die beiden benachbarten Dreiecke in eine Ebene fallen. Deshalb ist in der tatsächlichen Implementierung noch etwas extra Logik enthalten.

[2] Wie bereits im entsprechenden Kapitel angemerkt, spielt es für unsere Gleichungen keine Rolle, ob wir die z-Komponente auf 1 oder -1 normieren.

Literaturverzeichnis

[1] V.I. Oliker und L.D. Prussner. "On the Numerical Solution of the Equation ....-(....) = f and Its Discretizations, I." eng. In: *Numerische Mathematik* 54.3 (1989), S. 271–294 (siehe S. iii, 1–3, 9, 11, 12, 16, 20, 23, 25, 27–29, 35, 39).

[2] Alain Bossavit. *Computational Electromagnetism*. en. Electromagnetism. San Diego, CA: Academic Press, Feb. 1998 (siehe S. 1).

[3] Jos Thijssen. *Computational Physics*. 2. Aufl. Cambridge, England: Cambridge University Press, März 2007 (siehe S. 1).

[4] Franz Durst. *Grundlagen der Strömungsmechanik*. de. 2006. Aufl. Berlin, Germany: Springer, Mai 2006 (siehe S. 1).

[5] Paul Wilmott, Sam Howison und Jeff Dewynne. *The mathematics of financial derivatives: A student introduction*. Cambridge University Press, 2010 (siehe S. 1).

[6] Guido De Philippis und Alessio Figalli. *The Monge-Ampère equation and its link to optimal transportation*. 2013. arXiv: 1310.6167 [math.AP] (siehe S. 1).

[7] Alessio Figalli. *The Monge–Ampère Equation and Its Applications*. EMS Press, Jan. 2017 (siehe S. 1, 3–6).

[8] C.J. Budd, M.J.P. Cullen und E.J. Walsh. "Monge–Ampére based moving mesh methods for numerical weather prediction, with applications to the Eady problem". In: *Journal of Computational Physics* 236 (2013), S. 247–270 (siehe S. 1).

[9] Xu-Jia Wang. "On the design of a reflector antenna". In: *Inverse Problems* 12.3 (Juni 1996), S. 351 (siehe S. 1).

[10] Xu-Jia Wang. "On the design of a reflector antenna II". In: *Calculus of Variations and Partial Differential Equations* 20.3 (Juli 2004), S. 329–341 (siehe S. 1).

[11] Alfio Quarteroni, Riccardo Sacco und Fausto Saleri. *Numerical Mathematics*. Springer New York, 2007 (siehe S. 19).

[12] Raimund Seidel. "The upper bound theorem for polytopes: an easy proof of its asymptotic version". In: *Computational Geometry* 5.2 (1995), S. 115–116 (siehe S. 25).

[13] C. Bradford Barber, David P. Dobkin und Hannu Huhdanpaa. "The quickhull algorithm for convex hulls". In: *ACM Trans. Math. Softw.* 22.4 (Dez. 1996), S. 469–483 (siehe S. 25).

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich trage die Verantwortung für die Qualität des Textes sowie die Auswahl aller Inhalte und habe sichergestellt, dass Informationen und Argumente mit geeigneten wissenschaftlichen Quellen belegt bzw. gestützt werden. Die aus fremden oder auch eigenen, älteren Quellen wörtlich oder sinngemäß übernommenen Textstellen, Gedankengänge, Konzepte, Grafiken etc. in meinen Ausführungen habe ich als solche eindeutig gekennzeichnet und mit vollständigen Verweisen auf die jeweilige Quelle versehen. Alle weiteren Inhalte dieser Arbeit ohne entsprechende Verweise stammen im urheberrechtlichen Sinn von mir.

Ich weiß, dass meine Eigenständigkeitserklärung sich auch auf nicht zitierfähige, generierende KIAnwendungen (nachfolgend „generierende KI“) bezieht. Mir ist bewusst, dass die Verwendung von generierender KI unzulässig ist, sofern nicht deren Nutzung von der prüfenden Person ausdrücklich freigegeben wurde (Freigabeerklärung). Sofern eine Zulassung als Hilfsmittel erfolgt ist, versichere ich, dass ich mich generierender KI lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss deutlich überwiegt. Ich verantworte die Übernahme der von mir verwendeten maschinell generierten Passagen in meiner Arbeit vollumfänglich selbst. Für den Fall der Freigabe der Verwendung von generierender KI für die Erstellung der vorliegenden Arbeit wird eine Verwendung in einem gesonderten Anhang meiner Arbeit kenntlich gemacht. Dieser Anhang enthält eine Angabe oder eine detaillierte Dokumentation über die Verwendung generierender KI gemäß den Vorgaben in der Freigabeerklärung der prüfenden Person. Die Details zum Gebrauch generierender KI bei der Erstellung der vorliegenden Arbeit inklusive Art, Ziel und Umfang der Verwendung sowie die Art der Nachweispflicht habe ich der Freigabeerklärung der prüfenden Person entnommen.

Ich versichere des Weiteren, dass die vorliegende Arbeit bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt wurde oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen ist.

Mir ist bekannt, dass ein Verstoß gegen die vorbenannten Punkte prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass meine Prüfungsleistung als Täuschung und damit als mit „nicht bestanden“ bewertet werden kann. Bei mehrfachem oder schwerwiegendem Täuschungsversuch kann ich befristet oder sogar dauerhaft von der Erbringung weiterer Prüfungsleistungen in meinem Studiengang ausgeschlossen werden.

Ort, Datum	**Yannick Höffner**