M.Sc. Thesis
Master of Science in Angewandter Mathematik

Formoptimierung bei Variationsungleichungen und ihre Implementierung in FEniCS

Masterarbeit

zur Erlangung des akademischen Grades eines
Master of Science

vorgelegt am Fachbereich IV
der Universität Trier

von

Björn Führ

Erstgutachter: Prof. Dr. Volker Schulz
Zweitgutachter: Dr. Martin Siebenborn

Trier, 20. Oktober 2017

Universität Trier - Gebäude E
Fachbereich IV - Mathematik

Universitätsring 19
D-54296 Trier

Danksagung

An erster Stelle möchte ich mich bei allen bedanken, die mich während der Fertigstellung dieser Arbeit unterstützt haben und mir motivierend zur Seite standen.

Mein besonderer Dank gilt meinen Betreuern Prof. Dr. Volker Schulz und Dr. Kathrin Welker, mit denen ich jederzeit Kontakt aufnehmen und aufschlussreiche Diskussionen führen konnte. So wurde stets eine Lösung für auftretende Probleme gefunden und Unklarheiten konnten beseitigt werden. Ebenfalls möchte ich Dr. Martin Siebenborn für die Unterstützungen bei programmiertechnischen Fragestellungen danken.

Zu guter Letzt danke ich all meinen Freunden und Kommilitonen, ohne die meine Studienzeit nicht so unterhaltsam gewesen wäre.

Trier, 20. Oktober 2017

Björn Führ

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die moderne Mathematik ermöglicht eine Beschreibung vieler fächerübergreifender Probleme aus Industrie und Wirtschaft. Durch den nicht anhaltenden Fortschritt der Computerindustrie und immer mehr verfügbaren Ressourcen ist es möglich stets größere und komplexere Probleme zu lösen oder Simulationen zu berechnen. Viele dieser auftretenden Probleme aus der Natur und Technik können dabei mittels partiellen Differentialgleichungen (PDE) beschrieben werden. Diese tauchen somit auch im Gebiet der Formoptimierung auf, die in viele Anwendungsbereichen von großem Interesse ist. Die Form spielt zum Beispiel in der Luft- und Raumfahrt Industrie[1], aber auch in der Akkustik[2] oder sogar bei der Modellierung von Hautstrukturen[3] eine wichtige Rolle, um nur ein paar Beispiele zu nennen.

In dieser Arbeit beschäftigen wir uns mit der Formoptimierung bei Variationsungleichungen. Dazu bauen wir auf den Ergebnissen von [8, 9, 10] auf und bringen zusätzlich Variationsungleichungen mit ins Spiel. Ein gegebenes Zielfunktional soll minimiert werden, wobei ein System von partiellen Differentialgleichungen einzuhalten ist, in dem die Variationsungleichung eine Rolle spielt. Ein mögliches Minimierungsziel ist zum Beispiel der Abstand einer Lösung aus einer partiellen Differentialgleichung zu vorgegebenen Ziel- oder Messdaten. Die Lösung der PDE wird dabei durch Verformung des zugrundeliegenden Gitters an die gewünschten Messdaten angenähert, daher auch der Begriff Formoptimierung. Bei diesem Verfahren handelt es sich um ein Gradienten basiertes Verfahren, wobei es sich hier um den sogenannten Formgradienten handelt, der die Richtung der Deformation angibt. Da auch hier partielle Differentialgleichungen zu lösen sind, ist es möglich auf vorhandene Ergebnisse zurückzugreifen. Für die Lösung des Systems mit Variationsungleichungen bedienen wir uns am Verfahren aus [5], das an die Gegebenheiten in dieser Arbeit angepasst wird, unter anderem mit Hilfe von [4].

Ziel dieser Arbeit ist es ein Verfahren zu entwickeln und implementieren, das die Lösung einer Zustandsgleichung an vorhandene Messdaten annähert. Dies geschieht durch die Deformation des Gitters auf dem die Lösungen der Gleichungssysteme erzeugt werden. Die Lösungen befinden sich in einem quadratischen Gitter $(0,1)^2 \in \mathbb{R}^2$, in dem sich ein inneres Gebiet befindet, auf dem sich die Lösung der Differentialgleichung anders verhält. Um die Korrektheit des Verfahrens sicher zu stellen, werden die Messdaten als Lösung auf einem ausgewählten Gitter erzeugt. Dann wird auf einem anderen Gitter gestartet und überprüft, ob sich dieses zum vorher gewählten Zielgitter verformt. Als Programmiersprache wird Python verwendet, um den Algorithmus zu implementieren. Die Lösungen der Differentialgleichungen werden mit Hilfe der finiten Elemente Methode berechnet, bei der wir auf die open-source Bibliothek FEniCS zurückgreifen können. Diese ermöglicht es uns viele Modelle einfach in Python zu übertragen. Um die Ergebnisse zu visualisieren verwenden wir ParaView. Mit Hilfe dieses Programmes, ebenfalls open-source, ist es möglich die Gitterdateien sowie die Lösungen der Differentialgleichungssysteme zu visualisieren und analysieren. Der zeitliche Verlauf der Lösungen lässt sich sogar als Animation darstellen.

[1] S. Schmidt, C. Ilic, V.H. Schulz und N.R. Gauger. Three-dimensional largescale aerodynamic shape optimization based on shape calculus. AIAA Journal, 2013

[2] R. Udawalpola und M. Berggren. Optimization of an acoustic horn with respect to efficiency and directivity. International Journal for Numerical Methods in Engineering, 2008

[3] A. Nägel, V.H. Schulz, M. Siebenborn und G. Wittum. Scalable shape optimization methods for structured inverse modeling in 3D diffusive processes. Computing and Visualization in Science, 2015

Die Arbeit ist folgendermaßen aufgebaut:

Um den Einstieg zu erleichtern werden in **Kapitel 2** Grundlagen eingeführt, die für den späteren Verlauf wichtig sind. Zum einen werden generelle Voraussetzungen aus der Numerik der Differentialgleichungen beschrieben, wie die Definition der Poisson Gleichung, deren schwache Formulierung sowie den zur Herleitung notwendigen Satz von Green. Anschließend werden Begriffe und Notationen aus der Formoptimierung eingeführt, zu denen zum Beispiel die Form- und Materialableitung gehörigen. Der Satz von Green wird hier auf den zugrundeliegenden quadratischen Bereich mit innerem Teilgebiet übertragen.

In **Kapitel 3** definieren wir unser Ausgangsproblem mit entsprechendem Zielfunktional. Dort werden analog zu [8, 9, 10] neben der Zustandsgleichung, auch die adjungierten sowie lineare Elastizitätsgleichung beschrieben. Die Lösungen der Zustands- und adjungierten Gleichung werden dabei benutzt um die sogenannte Formableitung aufzustellen, die in der linearen Elastizitätsgleichung als rechte Seite verwendet wird, um die Deformation des Gitters zu berechnen. Bei der Lösung der Elastizitätsgleichung handelt es sich um ein Vektorfeld, das angibt in welche Richtung und wie stark sich jeder einzelne Knoten des Gitters verschiebt. Durch diese Deformation wird das Gitter verschoben, um sich den Messdaten anzunähern.

Die Implementierung in Python wird in **Kapitel 4** näher beschrieben. Das Verfahren wird unter dem Namen `SOVI-Toolbox` entwickelt. Zuerst wird hier auf die Erstellung der Gitterdateien eingegangen. Anschließend betrachten wir in eigenen Abschnitten die Implementierung der einzelnen Probleme aus Kapitel 3. Dazu wird ein Abschnitt des Quellcodes präsentiert und kommentiert. Abschließend ist eine Übersicht über den Aufbau der `SOVI-Toolbox` gegeben, in der die verschiedenen Funktionen zur Lösung der besprochenen Probleme aufgerufen werden. Es wird außerdem dargestellt, welche Dateien als Ausgabe erzeugt werden, um die berechneten Lösungen zum Beispiel mit dem Programm ParaView zu analysieren.

Die Ergebnisse von drei Testproblemen werden in **Kapitel 5** dargestellt. Einmal handelt es sich um die Verformung eines kleinen zentrierten Kreises zu einem größeren Kreis mit gleichem Mittelpunkt. In einem anderen Fall wird eine Ellipse zu einem Kreis geformt und im drittem und letzten Fall wird die Deformation eines zentrierten Kreises zu einem verschobenem Kreis mit anderem Mittelpunkt getestet. Die Deformationen des inneren Randes werden dabei im zeitlichen Verlauf dargestellt. Die Veränderung des Gitters wird in diesem Kapitel ebenfalls visualisiert.

Die Arbeit endet mit **Kapitel 6**, das eine abschließende Zusammenfassung und mögliche zukünftige Überlegungen liefert.

Kapitel 2

Grundlagen

Bevor wir uns mit der Formoptimierung bei Variationsungleichungen befassen, wollen wir erst noch die wichtigsten Grundlagen der Numerik bei Differentialgleichungen sowie Begriffe und Notationen der Formoptimierung einführen. Dazu zählen vor allem der Satz von Green, um die schwache Formulierung der partiellen Differentialgleichungssysteme herzuleiten, sowie im Bereich der Formoptimierung die Begriffe der Form- und Materialableitung.

Unser Problem ist in dieser Arbeit auf dem Einheitsquadrat in $\mathbb{R}^2$ definiert, wobei sich der gesamte Bereich Ω in ein inneres Teilgebiet Ω_2 und den Rest $\Omega_1 = \Omega \backslash \Omega_2$ aufteilt, siehe dazu Abbildung 2.1. Da wir uns später nur mit dieser Art von Gebieten befassen, sind im Folgenden manche Sätze und Definitionen explizit auf diesen Fall angepasst.

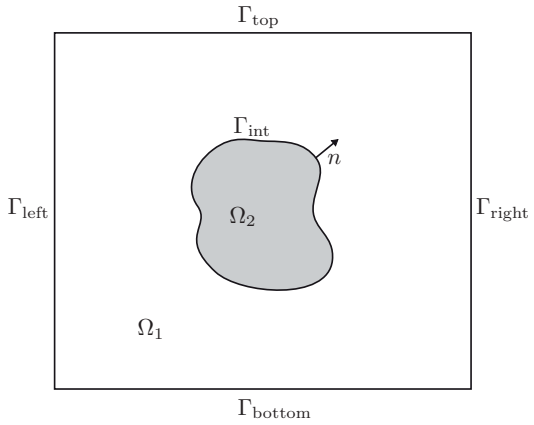


Abbildung 2.1: Gesamtes Gebiet Ω mit den einzelnen Bestandteilen Ω_1 und Ω_2. Der Rand von Ω_2 wird dabei mit Γ_{int} bezeichnet. Die Ränder des Einheitsquadrates erhalten die Bezeichnung entsprechend ihrer Ausrichtung, hier $\Gamma_{\text{left}}, \Gamma_{\text{top}}, \Gamma_{\text{right}}$ und Γ_{bottom} [9].

Wir befassen uns in dieser Arbeit mit der Problematik der Formoptimierung, weshalb vor allem der Rand Γ_{int} eine große Rolle spielt. Die Lösung der Differentialgleichung verhält sich auf den einzelnen Gebieten unterschiedlich. Aus diesem Grund bringt eine Deformation des inneren Randes eine andere Lösung mit sich. Die anderen vier äußeren Ränder sind hingegen nicht von Bedeutung. Hier wird an jeden eine Dirichlet-Randbedingung gestellt und das Verhalten dort somit a priori festgelegt.

2.1 Numerik der Differentialgleichungen

Im nächsten Kapitel beschäftigen wir uns mit dem eigentlichen Formoptimierungsproblem bei Variationsungleichungen. Im Verfahren tauchen verschiedene Probleme auf, die sehr verwandt mit dem Poisson Problem sind, weshalb wir es im Folgenden definieren:

DEFINITION 2.1 (vgl. [2]): *Für $y : \Omega \to \mathbb{R}$ und $f : \Omega \to \mathbb{R}$ heißt die Gleichung*

$$-\Delta y = f$$

Poisson-Gleichung und mit einer zusätzlichen Dirichlet-Randbedingung erhalten wir das Poisson-Problem

$$\begin{aligned} -\Delta y &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega, \end{aligned} \tag{2.1}$$

wobei wir hier die Lösung auf dem Rand Null setzen.

Wie in [14] suchen wir jedoch keine klassische Lösung $y \in \mathcal{C}^2(\Omega) \cap \mathcal{C}(\bar{\Omega})$ der Poissongleichung, sondern eine *schwache* Lösung $y \in H_0^1(\Omega)$, wobei wie üblich mit $H_0^1(\Omega)$ der Sobolev-Raum (auch Sobolew-Raum) bezeichnet wird. Eine genauere Ausführung zu Sobolev-Räumen ist in [14] zu finden.

Um die schwache Lösung zu erhalten, muss man das Problem erst als Variationsformulierung beziehungsweise schwache Formulierung umschreiben. Dabei wird uns der Integralsatz von Green eine große Hilfe sein. Durch Anwendung dieses Satzes verschwindet der Laplace-Operator und wir benötigen nur noch die einfache Differenzierbarkeit der Funktionen, jedoch spielt auch das Verhalten der Funktion auf dem Rand noch eine Rolle. Im Falle unserer Null-Randbedingung fallen die Integrale über den Rand weg. Wenn wir uns später in der Formoptimierung damit befassen, müssen wir jedoch den inneren Rand vom Gebiet Ω_2 betrachten, an den wir a priori keine Bedingung gestellt haben.

SATZ 2.2 (vgl. [2]): *Sei $\Omega \in \mathbb{R}^d$ beschränkt. Dann gilt für die skalaren Funktionen $y, v \in \mathcal{C}^2(\Omega)$:*

(a) $$\int_\Omega (-\Delta y)\, v \,\mathrm{d}x = \int_\Omega \nabla y^T \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \tfrac{\partial y}{\partial \vec{n}} v \,\mathrm{d}s$$

(b) $$\int_\Omega (-\Delta y)\, v \,\mathrm{d}x = \int_\Omega y(-\Delta v) \,\mathrm{d}x - \int_{\partial\Omega} \tfrac{\partial y}{\partial \vec{n}} v - y \tfrac{\partial v}{\partial \vec{n}} \,\mathrm{d}s$$

Jetzt wollen wir unter Zuhilfenahme von Satz 2.2 die schwache Formulierung unseres Poisson-Problems herleiten. Dazu multiplizieren wir die Gleichung (2.1) mit einer beliebigen, aber festen, Testfunktion $v \in H_0^1(\Omega)$ und erhalten

$$\int_\Omega (-\Delta y)\, v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x.$$

Anschließend wenden wir auf das linke Integral den Satz von Green an, hier Teil (a) von Satz 2.2, und es folgt

$$\int_\Omega \nabla y^T \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \tfrac{\partial y}{\partial \vec{n}} v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x.$$

Das Integral über den Rand fällt dabei weg, da wir hier Testfunktionen v aus dem Raum $H_0^1(\Omega)$ verwenden, die nach Definition auf dem Rand verschwinden. Wir können dadurch folgende Definition festhalten (vgl. [14]):

DEFINITION 2.3 (vgl. [14]): *Ein $y \in H_0^1(\Omega)$ heißt schwache Lösung des Poisson-Problems (siehe Definition 2.1), wenn die Variationsformulierung oder schwache Formulierung*

$$\int_\Omega \nabla y^T \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall\, v \in H_0^1(\Omega)$$

erfüllt ist.

2.2 Formoptimierung

Bevor wir uns dem Formoptimierungsproblem widmen, wollen wir im Folgenden die Terminologie und wichtige Aussagen aus dem Bereich Formoptimierung vorstellen. Dabei werden wir vor allem die, für uns später wichtigen, Form- und Materialableitung definieren und des Weiteren eine Formulierung des Satzes von Green angeben, der unser spezielles Gebiet mit dem inneren Rand berücksichtigt. Dies wird uns besonders beim Aufstellen der schwachen Formulierung helfen, wie wir es auch im vorherigen Abschnitt gesehen haben.

Wie der Name Formoptimierung vermuten lässt, spielen dabei Form-Funktionale (engl. *shape functionals*) eine große Rolle, weshalb wir sie am Anfang definieren wollen.

Definition 2.4 (vgl. [15]): *Sei D eine nichtleere Teilmenge des $\mathbb{R}^d$, wobei $d \in \mathbb{N}$. Weiter bezeichnen wir mit $\mathcal{A} \subset \{\Omega : \Omega \subset D\}$ eine Menge von Teilmengen von D. Ein Funktional*

$$J : \mathcal{A} \to \mathbb{R}, \Omega \mapsto J(\Omega)$$

wird Form-Funktional genannt.

Das zugehörige Formoptimierungsproblem ist wie folgt gegeben:

Definition 2.5 (vgl. [15]): *Ein Formoptimierungsproblem ist gegeben durch*

$$\min_{\Omega} J(\Omega)$$

wobei J ein Form-Funktional ist. Falls J von der Lösung einer partiellen Differentialgleichung abhängt, nennen wir das Problem auch PDE beschränkt.

Unser Ziel ist es einen Algorithmus zu entwickeln, der ein Ausgangsgebiet so verformt, dass sich die Lösung auf ihm den Messdaten annähert. Dabei sollen natürlich die Variationsungleichungen als Nebenbedingungen eingehalten werden. Um diese Verschiebung zu beschreiben, definieren wir den Begriff eines deformierten Gebietes.

Definition 2.6 (vgl. [8, 9]): *Sei $T_t : \Omega \to \Omega$, $x \mapsto T_t(x)$, $t \in (0, \delta)$ sei eine hinreichend glatte Familie von bijektiven Abbildungen. Dann ist das deformierte Gebiet Ω_t definiert als*

$$\Omega_t := T_t(\Omega) = \{T_t(x) : x \in \Omega\}.$$

Hier betrachten wir als Familie von bijektiven Abbildung, um das Gebiet zu deformieren, die sogenannte *perturbation of identity*, welche definiert ist als

$$T_t(\Omega)[V] = x + tV(x)$$

für das ausreichend glatte Vektorfeld V [siehe [8, 9]]. Eine andere Methode, um die Bewegung eines Punktes in $x \in \Omega$ zu beschreiben, ist die *velocity method*, worin der Fluss $F_t(x) := \xi(t, x)$ als Lösung des Anfangswertproblems

$$\begin{aligned} \frac{\mathrm{d}\xi(t,x)}{\mathrm{d}t} &= V(\xi(t,x)) \\ \xi(0,x) &= x, \end{aligned}$$

bestimmt wird [vgl. [8, 9]].

Jetzt sind wir in der Lage zu definieren, was wir uns unter einer Ableitung vorstellen - hier die Formableitung. Analog zu der gewöhnlichen wird die Formableitung ebenfalls als Grenzwert gebildet, mit dem Unterschied, dass es sich um eine Ableitung in Richtung eines Vektorfeldes V handelt. Formal halten wir das in folgender Definition fest:

Definition 2.7 (vgl. [8, 9]): *Für $\Omega \in D$, $V \in C^2\left(\mathbb{R}^d, \mathbb{R}^d\right)$ heißt die Abbildung $J : D \to \mathbb{R}$ formdifferenzierbar, falls der Grenzwert*

$$\mathrm{D}J(\Omega)[V] := \lim_{t \to 0^+} \frac{J(\Omega_t) - J(\Omega)}{t}$$

für alle V existiert und die Abbildung $\mathrm{D}J(\Omega)[V] : C^2\left(\mathbb{R}^d, \mathbb{R}^d\right) \to \mathbb{R}$, $V \mapsto \mathrm{D}J(\Omega)[V]$ linear und stetig ist. Dann heißt $\mathrm{D}J(\Omega)[V]$ Formableitung von J in Richtung V.

Die Formableitung kann dabei auf zwei äquivalenten Arten ausgedrückt werden (siehe [8]). Einmal als Integral über den Rand und auch als Integral über das Gebiet:

$$\mathrm{D}J_\Omega[V] := \int_\Omega G(x)V(x)\,\mathrm{d}x \quad \text{(Volumen-Formulierung)},$$

$$\mathrm{D}J_\Gamma[V] := \int_{\partial\Omega} g(s)V(s)^T n(s)\,\mathrm{d}s \quad \text{(Rand-Formulierung)},$$

wobei $\mathrm{D}J_\Omega[V] = \mathrm{D}J(\Omega)[V] = \mathrm{D}J_\Gamma[V]$ gilt, mit entsprechenden Funktionen G und g. Es ist laut [12] außerdem immer möglich die Formableitung als Randintegral auszudrücken. Die Herleitung dieses Randintegrals ist jedoch aufwändiger und benötigt ebenfalls mehr Arbeit bei der Implementierung, weshalb wir uns später nur auf die Volumen-Formulierung beschränken werden.

Als nächste Ableitung führen wir die punktweise Materialableitung einer Funktion $\varphi_t : \Omega_t \to \mathbb{R}^d$ ein. Sie beschreibt das Verhalten der Funktion am Punkt $x \in \Omega$ in Richtung $V(x)$. Die Funktion φ_t kann dabei als Restriktion der Funktion $\varphi : \bigcup_{t\in[0,T]}\{t\} \times \Omega_t$ auf $\{t\} \times \Omega_t$ angesehen werden (vgl. [3]). In der nächsten Definition fassen wir obige Gedanken und die zugehörige punktweise Formableitung einer entsprechenden Funktion φ am Punkt $x \in \Omega$ zusammen:

Definition 2.8 (vgl. [3, 9]): *Sei die Funktionenschar $\left\{\varphi_t : \Omega_t \to \mathbb{R}^d \,|\, t \le T\right\}$ differenzierbar in t mit $\varphi := \varphi_0$ und $x_t \in T_t(\Omega)$. Dann gelten folgende Definitionen:*

- *Die Materialableitung von φ an der Stelle x ist definiert als totale Ableitung*

$$\mathrm{D_m}(\varphi)(x) := \dot{\varphi}(x) := \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} (\varphi_t(x_t))$$

  und wird mit $\dot{\varphi}$ oder $\mathrm{D_m}(\varphi)$ symbolisch dargestellt.

- *Die Formableitung von φ an der Stelle x ist festgelegt als*

$$\mathrm{D}\varphi(x) = \varphi'(x) := \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} (\varphi_t(x))\,,$$

  und wird entsprechend verkürzt als φ' geschrieben.

Zwischen den beiden Ableitungen besteht folgender Zusammenhang:

Satz 2.9 (vgl. [3]): *Es gilt*

$$\dot{\varphi}(x) = \varphi'(x) + \nabla\varphi(x)^T V(x).$$

Beweis. Mit der Kettenregel folgt sofort (siehe [3])

$$\dot{\varphi}(x) = \left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} (\varphi_t(x_t)) = \varphi'(x) + \frac{\partial\varphi_0}{\partial x}\frac{\mathrm{d}x_t}{\mathrm{d}t}.$$

Außerdem gilt $\frac{\partial\varphi_0}{\partial x}(x) = \frac{\partial\varphi}{\partial x}(x) = \nabla\varphi(x)^T$ und mit der *perturbation of identity* folgt aus

$$x_t = T_t(x) = x + tV(x),$$

dass

$$\frac{\mathrm{d}x_t}{\mathrm{d}t} = V(x).$$

□

Unter bestimmten Glattheitseigenschaften lassen sich, wie in [3] dargestellt, die Form- und räumliche Ableitung vertauschen

$$\left(\frac{\partial \varphi}{\partial x_i}\right)' = \frac{\partial}{\partial x_i}(\varphi').$$

Für die Materialableitung und den Gradienten ist das jedoch nicht erlaubt ihre Reihenfolge einfach zu vertauschen, wie wir im nächsten Satz sehen werden. Um später besser mit der Materialableitung umgehen zu können und unsere Rechnungen zu vereinfachen, werden wir verschiedene wichtige Eigenschaften der Materialableitung in einem Satz zusammenfassen, welche in [1] zu finden sind.

Satz 2.10 (vgl. [1]): *Seien φ, ψ Funktionsscharen wie in Definition 2.8. Dann gilt*

(a)

$$\mathrm{D_m}(\varphi\,\psi) = \mathrm{D_m}(\varphi)\psi + \varphi \mathrm{D_m}(\psi)$$

(b)

$$\mathrm{D_m}(\nabla\varphi) = \nabla \mathrm{D_m}(\varphi) - \nabla V^T \nabla\varphi$$

(c)

$$\mathrm{D_m}(\nabla\varphi^T\,\nabla\psi) = \nabla \mathrm{D_m}(\varphi)^T \nabla\psi - \nabla\varphi^T\left(\nabla V + \nabla V^T\right)\nabla\psi + \nabla\varphi^T \nabla \mathrm{D_m}(\varphi)$$

Für die Materialableitung gilt dementsprechend auch die Produktregel, hier in Punkt (a) formuliert. In (b) sehen wir, wie oben erwähnt, dass sich die Materialableitung und der Gradient nicht einfach vertauschen lassen - ein zusätzlicher Term $\nabla V^T \nabla\varphi$ muss berücksichtigt werden. Der letzte Punkt ist eine Kombination aus den beiden vorherigen und wird uns später viel Arbeit abnehmen, da im Rahmen der schwachen Formulierung ein Term der Form $\nabla\varphi^T\,\nabla\psi$ vorkommt.

Bei der Herleitung der Formableitung benötigen wir im nächsten Kapitel folgende Aussage über die Ableitung eines Integrals über ein deformiertes Gebiet.

Satz 2.11 (vgl. [3]): *Betrachte die Funktionsschar $\left\{\varphi_t : \Omega_t \to \mathbb{R}^d \,|\, t \le T\right\}$ mit $\varphi := \varphi_0$. Dann gilt*

$$\left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \left(\int_{\Omega_t} \varphi_t(x_t)\,\mathrm{d}x\right) = \int_\Omega \dot\varphi(x) + \varphi(x)\,\mathrm{div}\left(V(x)\right)\,\mathrm{d}x$$

Zusätzlich benötigen wir noch die Ableitung eines Oberflächenintegrals, diesmal über einen deformierten Rand, der ähnlich zum deformierten Gebiet definiert ist (siehe [15]).

Satz 2.12 (vgl. [15]): *Betrachte die Funktionsschar $\left\{\varphi_t : \Omega_t \to \mathbb{R}^d \,|\, t \le T\right\}$ mit $\varphi := \varphi_0$. Dann gilt*

$$\left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \left(\int_{\Gamma_t} \varphi_t(x_t)\,\mathrm{d}x\right) = \int_\Gamma \dot\varphi(s) + \varphi(s)\,\mathrm{div}_\Gamma\left(V(x)\right)\,\mathrm{d}s,$$

wobei mit $\mathrm{div}_\Gamma(V)$ die tangentiale Divergenz von V bezeichnet wird und durch

$$\mathrm{div}_\Gamma(V) = \mathrm{div}(V) - n^T \frac{\partial V}{\partial n}$$

definiert ist.

Unser Problem ist auf dem Gebiet gemäß Abb. 2.1 formuliert. Dort wollen wir die Stetigkeit am Rand Γ_{int} explizit fordern. Die Lösung soll beim Übergang zwischen Ω_1 und Ω_2 keinen Sprung machen. Um dies in unserem Problem formulieren zu können benötigen wir folgende Definition:

Definition 2.13 (vgl. [10]): *Das Sprungsymbol $[\![\cdot]\!]$ beschreibt den Sprung an den Übergangsstellen in Γ_{int} (siehe Abb. 2.1) und ist definiert als*

$$[\![v]\!] := v_1 - v_2 \text{ mit } v_1 := v\big|_{\Omega_1} \text{ und } v_2 := v\big|_{\Omega_2},$$

wobei alles auf Γ_{int} betrachtet werden muss.

Für das Sprungsymbol können wir noch folgende Eigenschaften formulieren, wobei für uns besonders der zweite Fall interessant ist.

Satz 2.14 (vgl. [9]): *Für das Sprungsymbol $[\![\cdot]\!]$ gilt:*

(a)
$$[\![ab]\!] = [\![a]\!]b_1 + a_2[\![b]\!] = a_1[\![b]\!] + [\![a]\!]b_2$$

(b)
$$[\![ab]\!] = 0, \text{ falls } [\![a]\!] = [\![b]\!] = 0$$

Beweis. (a) Es gilt nach Definition $[\![a]\!] = a_1 - a_2$ und $[\![b]\!] = b_1 - b_2$. Wir multiplizieren $[\![a]\!]$ mit b_1 und $[\![b]\!]$ mit a_2 und erhalten

$$[\![a]\!]b_1 = (a_1 - a_2)b_1 = a_1b_1 - a_2b_1,$$
$$a_2[\![b]\!] = a_2(b_1 - b_2) = a_2b_1 - a_2b_2,$$

wobei das Ganze auf Γ_{int} betrachtet werden muss. Durch Aufsummieren der beiden Teile ergibt sich

$$[\![a]\!]b_1 + a_2[\![b]\!] = a_1b_1 - a_2b_1 + a_2b_1 - a_2b_2 = a_1b_1 - a_2b_2 = [\![ab]\!]$$

(b) Der zweite Fall folgt sofort aus dem ersten. Seien $[\![a]\!] = 0$ und $[\![b]\!] = 0$. Nach (a) gilt

$$[\![ab]\!] = [\![a]\!]b_1 + a_2[\![b]\!].$$

Das Einsetzen liefert dann die gewünschte Behauptung $[\![ab]\!] = 0$.

□

Zu Beginn dieses Kapitels haben wir uns bereits mit der schwachen Formulierung und dem Satz von Green (Satz 2.2) befasst. Wie schon mehrfach erwähnt, werden wir auch im nächsten Kapitel eine Variationsformulierung angeben und müssen dabei den inneren Rand Γ_{int} bei der Bildung der Randintegrale im Satz von Green berücksichtigen. Um einfacher auf die Lösung zurückzugreifen geben wir im nächsten Satz die für uns relevante Formulierung an:

Satz 2.15: *Sei $p \in H_0^1(\Omega)$ und die Gebiete und Ränder gemäß Abbildung 2.1 definiert. Dann gilt:*

(a)
$$\int_\Omega (-\Delta y)p \,\mathrm{d}x = \int_\Omega \nabla y^T \nabla p \,\mathrm{d}x - \int_{\Gamma_{int}} [\![\tfrac{\partial y}{\partial \vec{n}} p]\!] \,\mathrm{d}s$$

(b)
$$\int_\Omega (-\Delta y)p \,\mathrm{d}x = \int_\Omega y(-\Delta p) \,\mathrm{d}x - \int_{\Gamma_{int}} [\![\tfrac{\partial y}{\partial \vec{n}} p]\!] - [\![y \tfrac{\partial p}{\partial \vec{n}}]\!] \,\mathrm{d}s$$

Beweis. Wir beweisen hier nur Teil (a). Der zweite Teil folgt durch analoges Vorgehen bei Anwendung von Satz 2.2 (b). Es gilt $\Omega = \Omega_1 \cup \Omega_2$. Bevor wir den Satz von Green anwenden, teilen wir das Integral auf und erhalten

$$\int_\Omega (-\Delta y)p \,\mathrm{d}x = \int_{\Omega_1} (-\Delta y)p \,\mathrm{d}x + \int_{\Omega_2} (-\Delta y)p \,\mathrm{d}x.$$

Jetzt wenden wir Satz 2.2 (a) an und es folgt für die jeweiligen Integrale

$$\int_{\Omega_1} (-\Delta y)p \,\mathrm{d}x = \int_{\Omega_1} \nabla y^T \nabla p \,\mathrm{d}x - \int_{\Gamma_{\text{int}}} \frac{\partial p}{\partial \vec{n}} v \,\mathrm{d}s - \int_{\partial\Omega} \frac{\partial p}{\partial \vec{n}} v \,\mathrm{d}s$$

und

$$\int_{\Omega_2} (-\Delta y)p \,\mathrm{d}x = \int_{\Omega_2} \nabla y^T \nabla p \,\mathrm{d}x - \int_{\Gamma_{\text{int}}} \frac{\partial y}{\partial \vec{n}} p \,\mathrm{d}s.$$

Hier ist zu beachten, dass wir beim Gebiet Ω_1 den äußeren Rand des Quadrates, sowie den inneren Rand Γ_{int} zu berücksichtigen haben. Das Integral über den Rand $\partial\Omega$ fällt jedoch weg, da $p \in H_0^1(\Omega)$ und somit auf dem äußeren Rand verschwindet. Durch Aufsummieren und Anwenden der Definition des Sprungsymboles erhalten wir die gewünschte Aussage. □

Kapitel 3

Formoptimierung bei Variationsungleichungen

In diesem Kapitel führen wir unser Modellproblem ein, das durch Variationsungleichungen beschrieben wird. Unser Ziel ist es ein Verfahren zu entwickeln, analog zu [8], um ein gegebenes Zielfunktional zu minimieren. Die Lösung einer partiellen Differentialgleichung mit Variationsungleichung soll dabei einen möglichst kleinen Abstand zu gegebenen Messdaten haben. Durch Verformungen des zugrundeliegenden Gitters für die Lösung der Zustandsgleichung werden die Werte denen der vorgegebenen Zieldaten angenähert. Durch die Lösung der linearen Elastizitätsgleichung wird ein Deformationsvektorfeld berechnet, das die Richtung und den Betrag der Gitterdeformation angibt.

Die eigentliche Problemformulierung werden wir jedoch zuerst in eine äquivalente regularisierte Formulierung umwandeln, welche besser implementiert werden kann. Wie in [8] geben wir anschließend die adjungierte Gleichung dazu an, deren Lösung wir ebenfalls in der Formableitung benötigen. Danach sind wir in der Lage die Formableitung aufzustellen. Diese werden wir verwenden, um das Gebiet in die *richtige Richtung* zu Verformen. Das Vektorfeld für diesen Schritt erhalten wir aus der Lösung der linearen Elastizitätsgleichung, wobei hier als Quellterm die Formableitung eingesetzt wird. Schlussendlich erhalten wir unseren Algorithmus durch iteratives Anwenden der beschriebenen Schritte, wobei zur Steigerung der Stabilität und Geschwindigkeit noch weitere Maßnahmen unternommen werden können, wie lokal variierende Lamé-Parameter bei der Lösung der linearen Elastizitätsgleichung oder ein BFGS-Update, das in dieser Arbeit jedoch nicht weiter betrachtet wird.

3.1 Ausgangsproblem mit Variationsungleichung

Das Gebiet $\Omega = \Omega_1 \cup \Omega_2$ besteht wie in Abbildung 2.1 aus den Teilen Ω_1 und Ω_2. Für Ω wählen wir das quadratische Gebiet $(0,1)^2$ als Teilmenge von $\mathbb{R}^2$. Der äußere Rand $\Gamma_{\text{out}} := \Gamma_{\text{left}} \cup \Gamma_{\text{top}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}}$ ist unveränderlich und grenzt somit das Gebiet ein, auf dem wir optimieren.

Mit z werden die Messdaten bezeichnet, die approximiert werden sollen. Dies geschieht durch Verformungen des Gitters. Das zu optimierende Zielfunktional lautet

$$J(y, \Gamma_{\text{int}}) = j(y, \Gamma_{\text{int}}) + j_{\text{reg}}(\Gamma_{\text{int}}) \tag{3.1}$$

mit den zwei Teilen

$$j(y, \Gamma_{\text{int}}) = \frac{1}{2} \int_{\Omega(\Gamma_{\text{int}})} (y - z)^2 \, \mathrm{d}x \tag{3.2}$$

und

$$j_{\text{reg}}(\Gamma_{\text{int}}) = \nu \int_{\Gamma_{\text{int}}} 1 \, \mathrm{d}s, \tag{3.3}$$

wobei $\nu > 0$. Der erste Teil $j(y, \Gamma_{\text{int}})$ beschreibt den kleinste Quadrate Abstand der Zustandsvariable y zu den Messdaten z, den es zu minimieren gilt. Beim zweiten Term $j_{\text{reg}}(\Gamma_{\text{int}})$ handelt es sich um die sogenannte Perimeter-Regularisierung, die zur Wohlgestelltheit des Problems notwendig ist und hier ein Vielfaches der Länge des inneren Randes Γ_{int} angibt.

Als Nebenbedingungen ist folgendes System, bestehend aus einer partiellen Differentialgleichung mit Variationsungleichung, zu lösen:

$$\begin{aligned} -\Delta y + \lambda &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega \\ y &\leq \psi \quad \text{in } \Omega \\ \lambda &\geq 0 \quad \text{in } \Omega \\ \lambda(y-\psi) &= 0 \quad \text{in } \Omega \\ [\![y]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}} \\ [\![\tfrac{\partial y}{\partial \vec{n}}]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}}. \end{aligned} \tag{3.4}$$

Die Lösung erfüllt alle Gleichungen und Ungleichungen des Systems, ist daher vor allem durch die Funktion ψ nach oben beschränkt. Die Funktion

$$f(x) = \begin{cases} f_1(x) = const. & \text{, falls } x \in \Omega_1, \\ f_2(x) = const. & \text{, falls } x \in \Omega_2. \end{cases}$$

ist stückweise konstant auf den jeweiligen Teilgebieten. Mit den Sprungsymbolen

$$[\![y]\!] = 0, \quad [\![\tfrac{\partial y}{\partial \vec{n}}]\!] = 0$$

fordern wir explizit die Stetigkeit des Zustandes, sowie des Flusses auf dem inneren Rand Γ_{int} (vgl. [15]).

3.2 Regularisierte Zustandsgleichung

Um die Nichtlinearität der Formableitung zu umgehen, formulieren wir in (3.5) eine regularisierte Version von (3.4).

$$\begin{aligned} -\Delta y + \lambda_c &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega \\ [\![y]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}} \\ [\![\tfrac{\partial y}{\partial \vec{n}}]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}}, \end{aligned} \tag{3.5}$$

wobei

$$\lambda_c := \max\{0, \bar{\lambda} + c(y-\psi)\}^2 \tag{3.6}$$

mit $c > 0$ und $0 \leq \bar{\lambda} \in L^4(\Omega)$ fest.

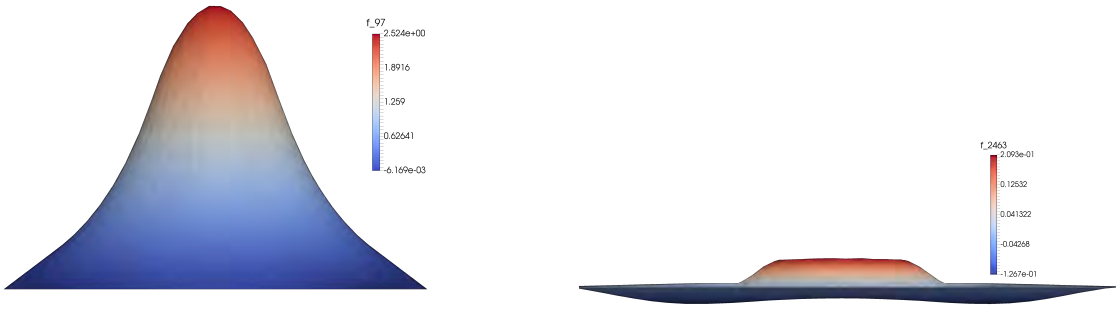
Zu diesem System wollen wir ebenfalls, wie schon im Grundlagen Kapitel, die schwache Formulierung aufstellen, die später im Computer implementiert wird. Dazu multiplizieren wir die Gleichung $-\Delta y + \lambda_c = f$ aus dem regularisierten System (3.5) mit einer beliebigen, aber festen Testfunktion $p \in H_0^1(\Omega)$ und integrieren über Ω. Dann erhalten wir

$$\int_\Omega (-\Delta y) p \,\mathrm{d}x + \int_\Omega \lambda_c p \,\mathrm{d}x = \int_\Omega f p \,\mathrm{d}x.$$

Auf das erste Integral wenden wir Satz 2.15 an und es ergibt sich insgesamt

$$\int_\Omega \nabla y^T \nabla p \,\mathrm{d}x = \int_\Omega f p \,\mathrm{d}x - \int_\Omega \lambda_c p \,\mathrm{d}x + \int_{\Gamma_{\text{int}}} [\![\tfrac{\partial y}{\partial \vec{n}} p]\!] \,\mathrm{d}s \tag{3.7}$$

als schwache Formulierung.



(a) ohne Beschränkung (b) mit Beschränkung $\psi = 0{,}2$

Abbildung 3.1: Lösungen der regularisierten Zustandsgleichung einmal mit und einmal ohne aktive Beschränkung. Hier wurden folgende Parameter gewählt: $c = 5$, $f_1 = -10$ und $f_2 = 100$. Links ist die unbeschränkte Lösung zu sehen, während rechts die Werte durch den überall konstanten Wert 0,2 beschränkt sind.

Eine Lösung der regularisierten Zustandsgleichung mit eingehaltener Variationsungleichung im Vergleich zu einer Lösung ohne Beschränkung ist in Abbildung 3.1 zu sehen. Ohne aktive Beschränkung gilt $\lambda = 0$ auf ganz Ω, wodurch es sich bei dem zu lösenden System um ein Poisson-Problem handelt, dessen Lösung in Abbildung 3.1(a) zu sehen ist. Wenn die Beschränkung ψ aktiv eingehalten wird, ist die resultierende Lösung nicht einfach an der Ebene abgeschnitten, hier in Abbildung 3.1(b) bei $\frac{1}{5}$, sondern das Verhalten über dem ganzen Gebiet Ω wird dadurch beeinflusst. Es treten sogar negative Werte auf im Vergleich zum Fall ohne Beschränkung. Die Lösungen der regularisierten Zustandsgleichung sind noch einmal in Abbildung 3.2 dargestellt. Dort ist wieder die Lösung ohne und mit Beschränkung zu finden. Zusätzlich ist in dieser Abbildung auch λ_c zu finden, das als Lösung des Systems (3.5) zusammen mit dem Zustand y berechnet wird. In diesem Plot ist deutlich zu sehen, wie sich die Lösung λ_c über dem Gebiet Ω verhält. Wie zu erwarten verschwindet die Funktion nach (4.1) dort, wo die Lösung y unterhalb der Schranke ψ ist und nimmt dort positive Werte an, wo die Grenze erreicht wird.

3.3 Lagrange-Funktion

Beim Aufstellen der Lagrange-Funktion betrachten wir die vereinfachte Zielfunktion ohne die Perimeter-Regularisierung j_{reg} zusammen mit der regularisierten Formulierung der Zustandsgleichung:

$$\min_{\Gamma_{\text{int}}} \frac{1}{2} \int_{\Omega(\Gamma_{\text{int}})} (y - z)^2 \, \mathrm{d}x$$

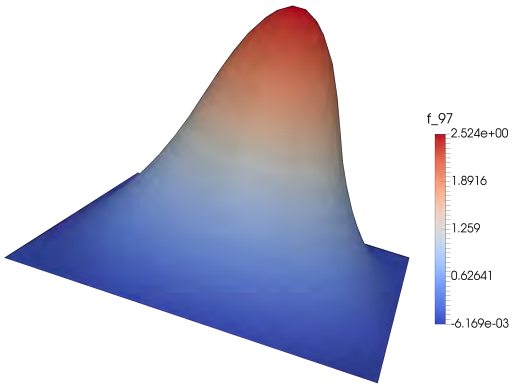
$$\begin{aligned} \text{s.d.} \quad -\Delta y + \lambda_c &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega \\ [\![y]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}} \\ [\![\tfrac{\partial y}{\partial \vec{n}}]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}}, \end{aligned}$$

wobei

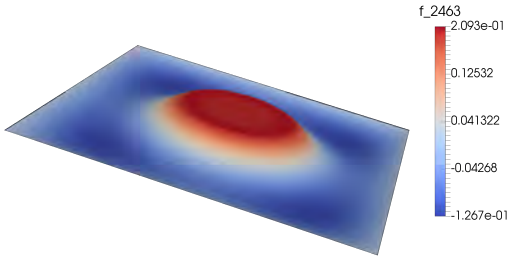
$$\lambda_c := \max\{0, \bar{\lambda} + c(y - \psi)\}^2$$

mit $c > 0$ und $0 \leq \bar{\lambda} \in L^4(\Omega)$ fest und

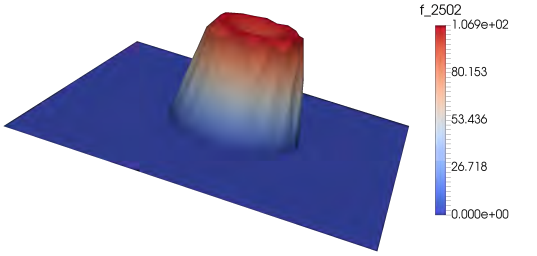
$$f(x) = \begin{cases} f_1(x) = const. & , \text{falls } x \in \Omega_1, \\ f_2(x) = const. & , \text{falls } x \in \Omega_2. \end{cases}$$



(a) Lösung der Zustandsgleichung ohne Beschränkung



(b) Lösung der Zustandsgleichung mit Beschränkung durch $\psi = 0{,}2$



s

(c) Zugehöriges λ_c um die Variationsungleichung einzuhalten in (b)

Abbildung 3.2: Lösungen der regularisierten Zustandsgleichung ohne und mit aktiver Beschränkung sowie dem dazugehörigen λ_c. Hier wurden folgende Parameter gewählt: $c = 5$, $f_1 = -10$ und $f_2 = 100$.

Dann lautet die Lagrangefunktion ohne den Regularisierungsterm j_{reg} bei analogem Vorgehen wie in [8, 9]:

$$\mathscr{L}(y, \Gamma_{\text{int}}, p) = \frac{1}{2} \int_\Omega (y - z)^2 \, \mathrm{d}x - \int_\Omega (\Delta y) p \, \mathrm{d}x - \int_\Omega f p \, \mathrm{d}x + \int_\Omega \lambda_c p \, \mathrm{d}x. \tag{3.8}$$

Auf den zweiten Term in (3.8) wenden wir Satz 2.2 (b) an und erhalten die finale Version der Lagran-

gefunktion

$$\begin{aligned}\mathscr{L}(y,\Gamma_{\text{int}},p) =& \frac{1}{2}\int_\Omega (y-z)^2 \,\mathrm{d}x - \int_\Omega y(\Delta p)\,\mathrm{d}x \\ & - \int_{\Gamma_{\text{int}}} [\![\tfrac{\partial y}{\partial \vec{n}} p]\!] - [\![y \tfrac{\partial p}{\partial \vec{n}}]\!]\,\mathrm{d}s - \int_\Omega fp\,\mathrm{d}x + \int_\Omega \lambda_c p\,\mathrm{d}x.\end{aligned} \tag{3.9}$$

Sie ist von elementarer Bedeutung, da aus ihr nicht nur, wie im nächsten Abschnitt das adjungierte Problem hergeleitet wird, sondern später auch Formableitung.

3.4 Adjungierte Differentialgleichung

Die adjungierte Differentialgleichung erlangen wir durch Ableiten der Lagrangefunktion nach y (vgl. [8])

$$\left.\frac{\mathrm{d}}{\mathrm{d}t}\right|_{t=0} \mathscr{L}(y+th,\Gamma_{\text{int}},p) \overset{!}{=} 0.$$

Mit Hilfe von [4] und [13] erhalten wir das sogenannte adjungierte Problem:

$$\begin{aligned} -\Delta p + 2c\sqrt{\lambda_c}p &= -(y-z) && \text{in } \Omega \\ p &= 0 && \text{auf } \partial\Omega \\ [\![p]\!] &= 0 && \text{auf } \Gamma_{\text{int}} \\ [\![\tfrac{\partial p}{\partial \vec{n}}]\!] &= 0 && \text{auf } \Gamma_{\text{int}}. \end{aligned} \tag{3.10}$$

Auch hier leiten wir die schwache Formulierung her, die später im Rechner implementiert wird. Dazu gehen wir analog, wie bei der Variationsformulierung der Zustandsgleichung vor. Ebenfalls kommt hier der Satz von Green 2.15 zur Anwendung. Sei dazu $v \in H_0^1(\Omega)$. Wir integrieren die adjungierte Gleichung über Ω und erhalten

$$\int_\Omega (-\Delta p)v\,\mathrm{d}x + \int_\Omega 2c\sqrt{\lambda_c}pv\,\mathrm{d}x = \int_\Omega -(y-z)v\,\mathrm{d}x$$

Jetzt wenden wir Satz 2.15 auf das erste Integral an und formen die Gleichung um und erhalten die schwache Formulierung des adjungierten Problems

$$\int_\Omega \nabla p^T \nabla v\,\mathrm{d}x + \int_\Omega 2c\sqrt{\lambda_c}pv\,\mathrm{d}x = \int_\Omega -(y-z)v\,\mathrm{d}x + \int_{\Gamma_{\text{int}}} [\![\tfrac{\partial p}{\partial \vec{n}} v]\!]\,\mathrm{d}s. \tag{3.11}$$

3.5 Formableitung

In diesem Abschnitt stellen wir die Formableitung auf, die für den Algorithmus von essentieller Bedeutung ist. Diese wird später als rechte Seite der linearen Elastizitätsgleichung implementiert, wodurch wir unser Vektorfeld erhalten, mit dessen Hilfe das Gitter in die entsprechende Richtung verformt wird. In der Formableitung tauchen alle bisher berechneten Variablen auf, wie der Zustand y, die Adjungierte p, λ_c und die rechte Seite der Zustandsgleichung f sowie die Messdaten z, die mittels der Verformungen approximiert werden sollen. Die Formableitung für das Zielfunktional ohne Perimeter-Regularisierung ist nach Anwendung von Satz 4.39 aus [15] gegeben durch

$$\begin{aligned}\mathrm{D}j(y,\Gamma_{\text{int}},p)[V] = \int_\Omega & -\nabla y^T(\nabla V + \nabla V^T)\nabla p - V^T \nabla f p \\ & + \operatorname{div}(V)\left[\tfrac{1}{2}(y-z)^2 + \nabla y^T \nabla p - fp + \lambda_c p\right]\,\mathrm{d}x\end{aligned} \tag{3.12}$$

Bei der Herleitung spielt für die Ableitung des Volumenintegrals Satz 2.11 und entsprechend für die Ableitung des Randintegrals Satz 2.12 eine wichtige Rolle. Die Formableitung ist die Ableitung der Lagrange-Funktion in Richtung eines Vektorfeldes V.

Die Formableitung der Perimeter-Regularisierung ist in [11] gegeben durch

$$\mathrm{D}j_{\mathrm{reg}}(\Gamma_{\mathrm{int}})[V] = \nu \int_{\Gamma_{\mathrm{int}}} \mathrm{div}(V) - \left(\frac{\partial V}{n}, n\right) \mathrm{d}s \tag{3.13}$$

Hier verwenden wir diese Version, die ohne den Krümmungsterm $\kappa := \mathrm{div}_{\Gamma_{\mathrm{int}}}(n)$ auskommt und daher einfacher zu implementieren ist.

3.6 Lineare Elastizitätsgleichung

Um mit Hilfe der Formableitung eine Verformung des Gitters zu erhalten, wird sie im Rahmen der linearen Elastizitätsgleichung als rechte Seite implementiert. Die lineare Elastizitätsgleichung ist wie in [8] von der Form

$$\begin{aligned} \mathrm{div}(\sigma) &= f_{\mathrm{elas}} && \text{in } \Omega \\ U &= 0 && \text{auf } \partial\Omega \end{aligned} \tag{3.14}$$

$$\begin{aligned} \sigma &:= \lambda_{\mathrm{elas}} \,\mathrm{Tr}(\epsilon) I + 2\mu_{\mathrm{elas}}\epsilon \\ \epsilon &:= \tfrac{1}{2}\left(\nabla U + \nabla U^T\right), \end{aligned}$$

wobei es sich bei σ um den Dehnungs- und bei ϵ um den Spannungstensor handelt. Hier beschreiben λ_{elas} und μ_{elas} die sogenannten Lamé-Parameter, die durch Youngs Elastizitätsmodul E und die Poissonzahl ν folgendermaßen dargestellt werden können

$$\lambda_{\mathrm{elas}} = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu_{\mathrm{elas}} = \frac{E}{2(1+\nu)}. \tag{3.15}$$

Hier haben sie direkt keine physikalische Bedeutung, sondern dienen zur Steuerung der Gitterverformung. Durch Youngs Elastizitätsmodul E können wir die Schrittweite steuern, während die Poissonzahl ν angibt, wie stark sich das Gitter in die übrigen Richtungen ausdehnt bei Ausübung einer Kraft in eine bestimmte Richtung (vgl. [8]).

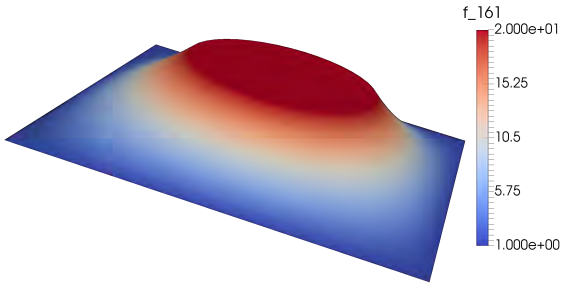
Analog wie in [7] ist das Ziel die Lamé-Parameter nicht über dem ganzen Gebiet konstant zu wählen, sondern lokal variieren zu lassen. Dadurch erhalten wir eine bessere Verformung des Gitters, bei der die Entstehung von allzu großen Elementen verringert werden kann. Wir wählen $\lambda_{\mathrm{elas}} = 0$ im gesamten Gebiet, wodurch sich der Dehnungstensor σ zu

$$\sigma = 2\mu_{\mathrm{elas}}\epsilon \tag{3.16}$$

vereinfacht. Der zweite Parameter μ_{elas} wird als Lösung folgenden Poisson Problems bestimmt

$$\begin{aligned} \Delta\mu_{\mathrm{elas}} &= 0 && \text{in } \Omega \\ \mu_{\mathrm{elas}} &= \mu_{\max} && \text{auf } \Gamma_{\mathrm{int}} \\ \mu_{\mathrm{elas}} &= \mu_{\min} && \text{auf } \partial\Omega. \end{aligned} \tag{3.17}$$

Bei unterschiedlicher Wahl von $\mu_{\min}$ und $\mu_{\max}$ erhält man somit eine lokal variierende Größe $\mu_{\mathrm{elas}}(x)$ mit $x \in \Omega$. Eine beispielhafte Lösung ist in Abbildung 3.3 dargestellt. Über die Wahl der Größen $\mu_{\min}$ und $\mu_{\max}$ lässt sich die Schrittweite und damit später auch die Laufzeit des Verfahrens beeinflussen. Je größer der Wert gewählt wird, desto kleiner sind die Schritte in jeder Iteration. Hier ist anzumerken, dass die Berechnung von μ_{elas} einmal am Anfang erfolgt und die Werte in jeder Iteration für weitere Berechnungen im Rahmen der linearen Elastizitätsgleichung benutzt werden. Es werden, um Rechenaufwand zu sparen, nicht in jeder Iteration neue Werte für μ_{elas} berechnet. Es ist außerdem auch noch möglich ein leicht verändertes Model für die Berechnung zu nehmen, in dem der Wert im inneren Gebiet Ω_2 nicht überall konstant ist, sondern nach innen hin abfällt. Dies wird hier aber nicht angewandt.



(a) Schräge Ansicht

(b) Seitliche Ansicht

Abbildung 3.3: Lokal variierende Lösung μ_{elas} des Problems (3.17). Auf dem inneren Rand Γ_{int} nimmt μ_{elas} den Wert $\mu_{\max}$ an, hier 20, und fällt zum äußeren Rand gemäß des Poisson Problem ab auf $\mu_{\min}$, hier 1. Im inneren Gebiet Ω_2 bleibt die Lösung konstant auf dem Wert $\mu_{\max}$, zu sehen in der schrägen Ansicht.

Unter Berücksichtigung des lokal variierenden Lamé-Parameters μ_{elas} und dem als konstant angenommenen zweiten Parameters λ erhalten wir mit (3.16) und der Formableitung als rechte Seite folgende lineare Elastizitätsgleichung

$$\begin{aligned} \operatorname{div}(\sigma) &= -\mathrm{D}j(y, \Gamma_{\text{int}}, p)[V] - \mathrm{D}j_{\text{reg}}(\Gamma_{\text{int}})[V] && \text{in } \Omega \\ U &= 0 && \text{auf } \partial\Omega \end{aligned} \tag{3.18}$$

$$\begin{aligned} \sigma &:= 2\,\mu_{\text{elas}}\,\epsilon \\ \epsilon &:= \tfrac{1}{2}\left(\nabla U + \nabla U^T\right). \end{aligned}$$

Die Lösung $U : \Omega \to \mathbb{R}^2$ wird anschließend, wie in [8], auf die Koordinaten der finite Elemente Knotenpunkte des zugrundeliegenden Gitters addiert, wodurch wir ein neues Gitter erhalten, auf dem in der nächsten Iteration die zuvor beschriebenen Probleme gelöst werden. Eine Lösung ist in Abbildung 3.4 dargestellt. Dort ist zu sehen, dass die Deformation am inneren Rand Γ_{int} (in grün markiert) am größten ist. Nach außen und weiter nach innen nimmt das Ausmaß der Deformation ab. Um mögliche Diskretisierungsfehler auszuschließen, wird später die Formableitung als rechte Seite der linearen Elastizitätsgleichung vor der Berechnung des Deformationsvektorfeldes modifiziert. Nach Assemblierung der Formableitung werden alle Werte, die zu Knoten des finiten Elemente Gitters gehören und keinen Träger am inneren Rand Γ_{int} haben, auf 0 gesetzt. Nur kleine Abweichung von 0 können schon schlimme Verformungen des Gitters zur Folge haben, wie in der aus [8] entnommenen Abbildung 3.5 dargestellt.

Die Modifikation der rechten Seite ist zu rechtfertigen, da es bei der Formableitung zwei äquivalente Formulierungen gibt. Einmal die hier angewendete Volumenformulierung und die Rand-Formulierung. Bei letzter wird das Integral nur über den inneren Rand Γ_{int} betrachtet, weshalb durch die Gleichwer-

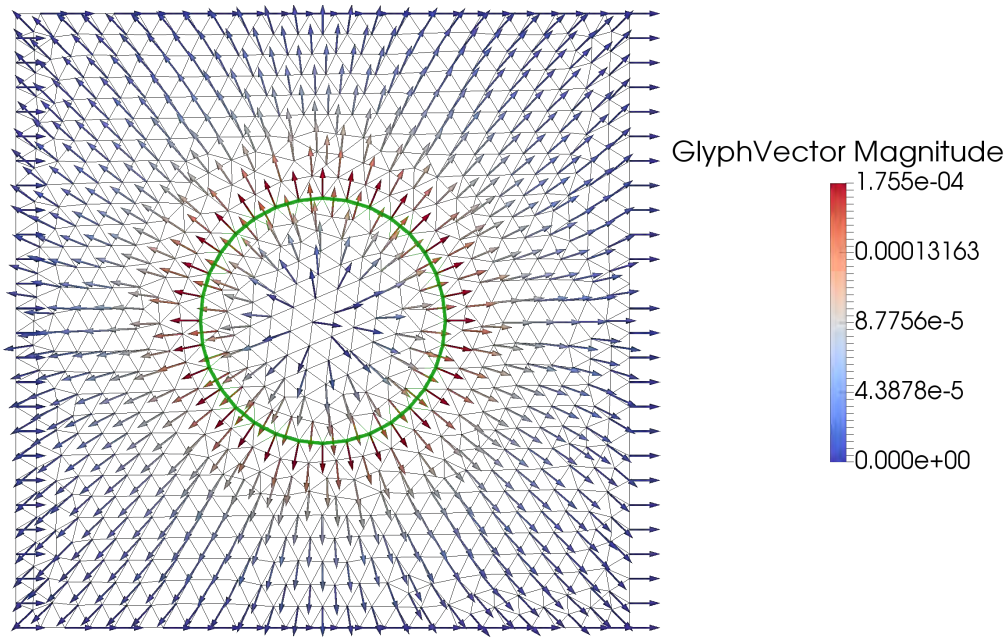


Abbildung 3.4: Deformationsvektorfeld U als Lösung der linearen Elastizitätsgleichung. In grün ist der Rand Γ_{int} des inneren Gebiets Ω_2 dargestellt, an dem die Verformung am stärksten ist. Zum äußeren Rand $\partial\Omega$ sowie zum Inneren von Ω_2 nimmt das Ausmaß der Verformung ab. Die Pfeile symbolisieren die Richtung der Verformung, während die Farbe die Länge der einzelnen Vektoren darstellt und damit die Größenordnung der Verformung. Diese ist anhand der Farbskala abzulesen.

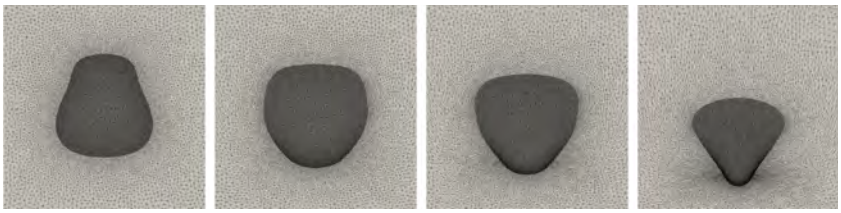
tigkeit beider Formulierungen nur die finiten Elemente eine Rolle spielen sollten, die sich am Rand Γ_{int} befinden.

3.7 Formoptimierungs Algorithmus

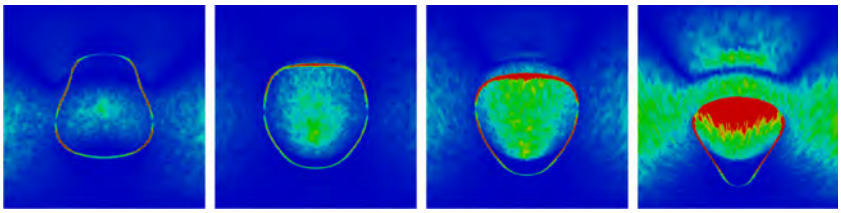
Zum Abschluss dieses Kapitels entwickeln wir, analog zu [8], eine Algorithmus zum Lösen des Formoptimierungsproblems mit Variationsungleichung. Als Eingabewerte wird neben den Parametern f, ψ, die das Optimierungsproblem beschreiben, ein Gitter auf dem Quadrat $(0,1)^2$ benötigt, worin sich separat gekennzeichnetes Gebiet Ω_2 befindet, welches verformt werden soll. Um das Ziel der Verformung festzulegen, sind Messdaten erforderlich. Um die Ergebnisse des Verfahrens validieren zu können, berechnen wir die Messdaten als Lösung der regularisierten Zustandsgleichung (3.5)

$$\begin{aligned} -\Delta y + \lambda_c &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega \\ [\![y]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}} \\ [\![\tfrac{\partial y}{\partial \vec{n}}]\!] &= 0 \quad \text{auf } \Gamma_{\text{int}}, \end{aligned}$$

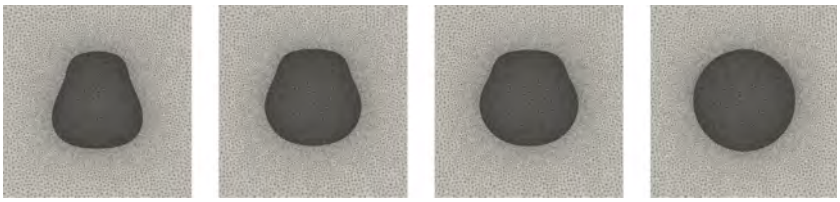
mit den selben Parametern, die auch das zu optimierende Problem beschreiben, jedoch mit einem unterschiedlichen inneren Gebiet Ω_2. Diese andere Form wird somit als Ziel für die Verformungen festgelegt. Bevor die Lösung als Messdaten dem Algorithmus als Input übergeben wird, werden die Werte noch durch normalverteilte Werte gestört.



(a) Iterationen ohne modifizierte rechte Seite



(b) Betrag der nicht modifizierten rechten Seite



(c) Richtige Deformation und Konvergenz auf Grund der modifizierten rechten Seite

Abbildung 3.5: Die Deformationen ohne modifizierte rechte Seite führen nicht zur optimalen Form, wie in (a) zu sehen ist. Das liegt daran, dass der Betrag der rechten Seite auch für finite Elemente Knoten, die keinen Träger auf dem inneren Rand besitzen, von 0 verschieden ist, wodurch die Verformungen gravierend beeinflusst werden, siehe (b). Im Vergleich dazu ist in (c) die reibungslose Konvergenz zur optimalen Form zu sehen. Hier wurde die rechte Seite entsprechend angepasst (vgl. [8]).

Ein Optimierungsschritt lässt sich wie folgt formulieren:

1. Interpolation der Zieldaten z auf das aktuelle Gitter
2. Lösen der Zustandsgleichung
3. Lösen der Adjungiertengleichung
4. Lösen der linearen Elastizitätsgleichung
5. Verschieben des aktuellen Gitters anhand der Lösung aus Schritt 4

Im Abschnitt der linearen Elastizitätsgleichung haben wir schon eine mögliche Verbesserung des Verfahrens beschrieben. Durch einen lokal variierenden Lamé-Parameter erhalten wir eine bessere Gitterkonvergenz und können durch die Wahl der minimalen und maximalen Werte auch Einfluss auf die Schrittweite ausüben. Als weiteren Schritt wäre zur Schrittweiten Kontrolle auch ein BFGS-Update denkbar, wie in [9]. Da die Anordnung der Gitterpunkte zueinander sich durch die Verforungen nicht verändert, dass heißt benachbarte Knotenpunkte bleiben benachbart und die Verbindungen überlappen sich nicht, reicht es aus, das Problem 3.17 einmal am Anfang zu lösen, um Rechenzeit zu sparen.

Den kompletten Algorithmus können wir somit wie folgt beschreiben:

Formoptimierungsverfahren bei Variationsungleichungen

1. Wähle oder generiere Zieldaten z, wähle Parameter f, ψ, $\mu_{\min}$ und $\mu_{\max}$ sowie Ω
2. Löse die Poisson Gleichung für den lokal variierenden Lamé-Parameter μ_{elas}
3. Interpoliere die Zieldaten z sowie den Lamé-Parameter μ_{elas} auf das aktuelle Gitter
4. Lösen der Zustandsgleichung
5. Lösen der Adjungiertengleichung
6. Lösen der linearen Elastizitätsgleichung
7. Verschieben des aktuellen Gitters anhand der Lösung aus Schritt 6
8. Stop oder gehe zu (3)

Diese Schritte werden im nächsten Kapitel im Rahmen der `SOVI-Toolbox` *(Shape Optimization with Variational Inequalities)* in der Programmiersprache Python implementiert, wobei das open-source Projekt FEniCS verwendet wird, das eine einfache Implementierung der Variationsformulierung mit finiten Elementen ermöglicht.

Kapitel 4

Implementierung in Python mit FEniCS

In diesem Kapitel gehen wir näher darauf ein, wie das im vorherigen Abschnitt erarbeitete Verfahren auf dem Computer umgesetzt werden kann. Dafür verwenden wir die Programmiersprache Python[1] zusammen mit der open-source Bibliothek FEniCS[2]. Das FEniCS Paket ermöglicht es uns, die schwachen Formulierungen sehr einfach durch eine Vielzahl an bereits vorhandenen Funktionen und Klassen am Computer zu implementieren, wobei auf [6] als Nachschlagewerk zurückgegriffen werden kann. Die Gitter wurden mit dem Programm Gmsh[3] erzeugt. Auf weitere Einzelheiten bei der Gittererstellung und wie diese später im Programmcode implementiert werden, wird im nächsten Abschnitt genauer eingegangen, bevor die verschiedenen Probleme aus dem vorherigen Kapitel in Python und FEniCS umgesetzt werden. Dazu zählen die regularisierte Formulierung, die adjungierte PDE und die lineare Elastizitätsgleichung zusammen mit der Berechnung der lokal variierenden Lamé-Parameter. Diese werden jeweils abschnittsweise detaillierter beschrieben. Dazu wird der Quellcode Stück für Stück betrachtet und kommentiert. Die einzelnen Abschnitte werden dann zum Schluss dieses Kapitels zu einer Toolbox zum Lösen von Formoptimierungsproblemen bei Variationsungleichungen zusammengefasst. Diese wird mit dem Namen

`SOVI-Toolbox`

bezeichnet, wobei die Abkürzung `SOVI` für *Shape Optimization with Variational Inequalities* steht. Der Quellcode zum kompletten Verfahren sowie die entsprechenden Dateien für die Gitter sind auf der beiliegenden CD gespeichert. Voraussetzung für die Nutzung sind lediglich eine funktionierende Python Installation mit dem Paket FEniCS, wobei die Lauffähigkeit nur für die am Anfang des Kapitels angegebenen Versionen garantiert werden kann.

Zuerst beginnen wir mit der Erstellung der Gitter und gehen darauf an wie diese im Programm verwendet werden:

4.1 Gittererzeugung

Hier wird auf die Erstellung der Gitter mit dem Programm Gmsh eingegangen. Dies kann mit Hilfe der grafischen Oberfläche des Tools oder auch in einem Texteditor geschehen. Gmsh Dateien haben die Endung `.geo`. Die ausgewählten Aktionen in der GUI von Gmsh werden automatisch in dazugehörige `.geo`-Datei geschrieben und können dort auch in einem Texteditor überarbeitet werden. An einem Beispiel gehen wir näher auf die Syntax in Gmsh ein. Wir wollen ein Gitter erstellen, das für unser Modell verwendet werden kann. Wir erzeugen hier ein quadratisches Gebiet mit einem inneren Kreis um den Mittelpunkt $(0{,}5\ 0{,}5)^T$ mit dem Radius 0,25.

Die Punkte setzen sich aus den x-,y- und z-Koordinaten zusammen, mit der charakteristischen Länge (engl. *characteristic length*) als vierte Größe, die die Elementgröße an diesem Punkt festlegt. Die ersten

[1] Version 2.7.1
[2] Version 2016.2.0
[3] Version 2.10.1

vier Punkte beschreiben das Quadrat auf der x-y-Ebene mit den Eckpunkten $(0,0),(1,0),(1,1),(0,1)$. Die Punkte $5,6$ und 7 werden benötigt, um einen Kreis zu definieren. Dabei ist der fünfte Punkt der Mittelpunkt und die anderen beiden beschreiben die jeweiligen Anfangs- und Endpunkte der Halbkreise, aus denen sich der innere Kreis zusammensetzt.

```
Point(1) = {0.0, 0.0,  0.0, 1.0};
Point(2) = {1.0, 0.0,  0.0, 1.0};
Point(3) = {1.0, 1.0,  0.0, 1.0};
Point(4) = {0.0, 1.0,  0.0, 1.0};
Point(5) = {0.5, 0.5,  0.0, 1.0};
Point(6) = {0.5, 0.25, 0.0, 1.0};
Point(7) = {0.5, 0.75, 0.0, 1.0};
```

Der äußere Rand des Quadrates wird durch Verbinden der vier Eckpunkte definiert.

```
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};
```

Der innere Kreis wird durch zwei Halbkreise mit dem Befehl `Circle` beschrieben. Der erste Wert bestimmt den Anfangspunkt, der zweite das Zentrum und der letzte den Endpunkt des Halbkreises. Das entspricht den *Linien* 5 und 6.

```
Circle(5) = {6, 5, 7};
Circle(6) = {7, 5, 6};
```

Um die einzelnen Teilgebiete festlegen zu können, müssen vorher sogenannte `Line Loop` Objekte definiert werden. Diese fassen die in den Klammern stehenden `Lines` zusammen. Der erste beschreibt das Quadrat, während der zweite den Kreis definiert.

```
Line Loop(1) = {1, 2, 3, 4};
Line Loop(2) = {6, 5};
```

Die Teilgebiete werden mit dem Befehl `Plane Surface` festgelegt. In den Klammern steht als erster Wert der äußere Rand des Teilgebiets und an zweiter Stelle der innere Rand. Für das zweite Teilgebiet wird nur ein Wert benötigt, da der Kreis keinen inneren Rand besitzt. Die Zahlen stehen für die Nummer des jeweiligen `Line Loop`, der dazu definiert wurde.

```
Plane Surface(1) = {1, 2};
Plane Surface(2) = {2};
```

Um später den vier äußeren sowie dem inneren Rand Werte zuzuweisen, werden diese zusätzlich als `Physical Line` implementiert.

```
Physical Line(1) = {1};
Physical Line(2) = {2};
Physical Line(3) = {3};
Physical Line(4) = {4};
Physical Line(5) = {5};
Physical Line(6) = {6};
```

Analog gilt das selbe für die Teilgebiete. Hier wird aus dem jeweiligen `Plane Surface` ein `Physical Surface` erstellt.

```
Physical Surface(1) = {1};
Physical Surface(2) = {2};
```

Um aus einer `name.geo`-Datei eine Gitterdatei zu generieren wird der Befehl

```
gmsh name.geo -2 -clscale 0.1
```

verwendet. Dazu wird der Befehl `gmsh` auf die entsprechende Datei `name.geo` mit den Parametern `-2` für ein 2D Gitter angewendet. Die Zahl hinter `-clscale` bestimmt die Gitterfeinheit. Als Resultat wird eine Gitterdatei mit der Endung `.msh` erstellt, die noch weiter konvertiert werden muss, um die Daten des Gitters in FEniCS zu laden. Dies erfolgt mit dem Befehl

```
dolfin-convert filename.msh filename.xml,
```

der die Datei in eine oder mehrere `.xml`-Dateien konvertiert. Dabei wird für die Zugehörigkeit der Knoten im Gitter zu den jeweiligen Teilgebieten eine Datei mit der Endung `_physical_region.xml` erstellt. Für die Ränder generiert der Konvertierungsbefehl eine separate Datei, die auf `_facet_region.xml` endet. Um diese Dateien zu erhalten, mussten wir die Teilgebiete und Ränder als `Physical` in der `.geo`-Datei definieren. In diesen Dateien werden den einzelnen Gebieten oder Rändern Indizes zugewiesen, um sie später im Programm einzeln anzusprechen. Für die Gebiete, die als `subdomains` (engl. für Teilgebiete) bezeichnet werden, ist eine mögliche Zuweisung in Abbildung 4.1 veranschaulicht. Für die inneren und äußeren Ränder ist das Schema analog. Die 4 äußeren Ränder bekommen die Indizes 1 bis 4. Dem inneren Rand werden, je nachdem aus wie vielen Linien er in Gmsh erstellt wurde, zusätzliche Indizes zugewiesen, entsprechend den Zahlen aus der `.geo`-Datei.

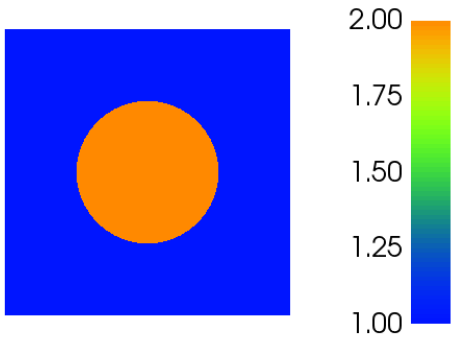


Abbildung 4.1: Den einzelnen Gebieten (`subdomains`) werden verschiedene Indizes zugewiesen. Das innere Gebiet Ω_2 bekommt den Index 2 (in orange), während das Gebiet Ω_1 den Wert 1 (in blau) zugewiesen bekommt.

Um im Programm alle Daten zum Gitter unter einem Namen zusammenzufassen, wird in Python die Klasse `MeshData` erstellt, die folgendermaßen aufgebaut ist:

```
class MeshData:
  def __init__(self, mesh, subdomains, boundaries, ind):
        # FEniCS Mesh
        self.mesh = mesh

        # FEniCS Subdomains
        self.subdomains = subdomains

        # FEniCS Boundaries
        self.boundaries = boundaries

        # Indizes der Knotenpunkte, die nicht auf dem inneren Rand liegen
        self.indNotIntBoundary = ind
```

In einem Objekt der Klasse `MeshData` sind unter `.mesh` alle Knotenpunkte unseres Gitters gespeichert, die benutzt werden um den zugrundeliegenden Raum zu definieren. In `.subdomains` wird die Zugehörigkeit der Knotenpunkte zum jeweiligen Teilgebiet festgelegt, siehe auch Abbildung 4.1. Unter `.boundaries` ist dementsprechend die Kennzeichnung der Punkte zu den Rändern definiert. Die

Indizes der Knotenpunkte, die nicht zum inneren Rand gehören, werden zusätzlich gespeichert, um deren Einfluss auf die Deformation zu verhindern. Sie werden unter dem Namen `.indNotIntBoundary` gespeichert.

Um später auf die Informationen des Gitters im Programmcode zugreifen zu können, befassen wir uns mit dem Einlesen der Daten, für die wir zur Speicherung das passende Objekt `MeshData` vorbereitet haben. Die dafür vorgesehene Funktion ist `load_mesh(name)`:

```
def load_mesh(name):

  # Pfad zur speziellen Gitter Datei
  path_meshFile = os.path.join(DATA_DIR, name)

  # Path to mesh file
  path_meshFile = os.path.join(DATA_DIR, name)

  # Create mesh and define function space
  mesh = Mesh(path_meshFile + ".xml")
  subdomains = MeshFunction("size_t", mesh, path_meshFile + "_physical_region.xml")
  boundaries = MeshFunction("size_t", mesh, path_meshFile + "_facet_region.xml")
  ind = __get_index_not_interior_boundary(mesh, subdomains, boundaries)

  return MeshData(mesh, subdomains, boundaries, ind)
```

Sie benötigt als Eingabe einen Namen für die Datei, unter dem das, ins `.xml`-Format konvertierte, Gitter gespeichert wurde. Dabei wird die Endung beim Namen weggelassen, da sie vom Programm automatisch ergänzt wird, ebenso wie die zum Gitter gehörenden zusätzlichen Dateien, die Teilgebiete und Ränder definieren. Alle drei `.xml`-Dateien müssen dazu im Ordner `Meshes` in der `SOVI Toolbox` abgelegt sein. Zuerst wird die Hauptdatei `filename.xml` als Gitter in FEniCS importiert. Mit Hilfe der Funktion `MeshFunction` und den weiteren `.xml`-Dateien werden den Knotenindizes des bereits importierten Gitters die richtige Nummer des jeweiligen Teilgebietes oder Randes zugeordnet und in den Variablen `subdomains` und `boundaries` gespeichert, bevor alle in einem `MeshData` Objekt zusammengefasst werden, welches der Rückgabewert der Funktion ist.

Die Gitterdaten können jetzt verwendet werden, um auf ihnen die partiellen Differentialgleichungen zu lösen. Wir gehen auf die Implementierung abschnittsweise ein und starten mit der regularisierten Gleichung.

4.2 Regularisierte Gleichung

Hier befassen wir uns mit der Implementierung der regularisierten Formulierung, die folgendermaßen gegeben war:

$$\begin{aligned} -\Delta y + \lambda_c &= f \quad \text{in } \Omega \\ y &= 0 \quad \text{auf } \partial\Omega \\ [\![y]\!] &= 0 \quad \text{auf } S \\ [\![\tfrac{\partial y}{\partial \vec{n}}]\!] &= 0 \quad \text{auf } S \end{aligned}$$

wobei

$$\lambda_c := \max\{0, \bar{\lambda} + c(y - \psi)\}^2 \quad \text{mit } c > 0 \text{ und } 0 \leq \bar{\lambda} \in L^4(\Omega) \text{ fest.}$$

Die schwache Formulierung zu diesem Problem war in (3.7) gegeben durch

$$\int_\Omega \nabla y^T \nabla p \, \mathrm{d}x = \int_\Omega f p \, \mathrm{d}x - \int_\Omega \lambda_c p \, \mathrm{d}x + \int_S [\![\tfrac{\partial y}{\partial \vec{n}} p]\!] \, \mathrm{d}s.$$

Um dieses Problem zu lösen, adaptieren wir den *primal-dual active set (PDAS)* Algorithmus aus [5]. Dort wird ein ähnliches regularisiertes Problem gelöst. Die Lösung des dort beschriebenen und die

Lösung des hier dargestellten Problems konvergieren nach [4] gegen das selbe Ergebnis. In [5] wird der PDAS Algorithmus auf folgende Weise beschrieben, wobei hier die einzelnen Schritte an unser Problem angepasst wurden:

Modified Primal-dual active set (mPDAS) Algorithmus

1. Wähle y_0, setze $k = 0$ und $\lambda_0 = 0$
2. Setze $\mathcal{A}_{k+1} = \{x : (\lambda_k + c(y - \psi))(x) > 0\}$ und $\mathcal{I}_{k+1} = \Omega \backslash \mathcal{A}_{k+1}$
3. Setze die Lösung folgender Gleichung als $y_{k+1} \in H_0^1(\Omega)$:
$$a(y, v) + ([\lambda_k + c(y - \psi)]^2, \chi_{\mathcal{A}_{k+1}} v) = (f, v) \quad \forall\, v \in H_0^1(\Omega) \tag{4.1}$$
4. Setze
$$\lambda_{k+1} = \begin{cases} 0 & , \text{falls } x \in \Omega_1, \\ \lambda_k + c(y_{k+1} - \psi) & , \text{falls } x \in \Omega_2. \end{cases}$$
5. Stop oder setze $k = k + 1$ und gehe zu (2)

Im dritten Schritt muss ein nichtlineares Problem in (4.1) gelöst werden. Um dies zu vermeiden lösen wir nicht direkt nach y_{k+1}, sondern nach Δy und erhalten unsere neue iterierte Lösung durch das Update

$$y_{k+1} = y_k + \Delta y. \tag{4.2}$$

Auf diese Weise erhalten wir aus (4.1) die Gleichung

$$a(y_k + \Delta y, v) + ([\lambda_k + c(y_k + \Delta y - \psi)]^2, \chi_{\mathcal{A}_{k+1}} v) = (f, v) \tag{4.3}$$

für alle $v \in H_0^1(\Omega)$. Durch das Quadrat ist der zweite Term noch immer nichtlinear, was durch folgende Linearisierung behoben werden kann:

$$(\lambda_k + c(y_k + \Delta y - \psi))^2 = (\lambda_k + c(y_k - \psi))^2 + 2c\Delta y(\lambda_k + c(y_k - \psi)).$$

Einsetzen in (4.3) und Umsortieren liefert

$$a(\Delta y, v) + (2c\Delta y(\lambda_k + c(y_k - \psi)), \chi_{\mathcal{A}_{k+1}} v) = (f, v) - a(y, v) - ([\lambda_k + c(y_k - \psi)]^2, \chi_{\mathcal{A}_{k+1}} v), \tag{4.4}$$

wodurch wir eine in Δy lineare Form von (4.1) erhalten. Mit dieser Umformulierung können wir eine lineare Version des mPDAS Algorithmus angeben, in dem wir Schritt 3 in zwei Teile gliedern.

Linearized modified Primal-dual active set (lmPDAS) Algorithmus

1. Wähle y_0, setze $k = 0$ und $\lambda_0 = 0$
2. Setze $\mathcal{A}_{k+1} = \{x : (\lambda_k + c(y - \psi))(x) > 0\}$ und $\mathcal{I}_{k+1} = \Omega \backslash \mathcal{A}_{k+1}$
3. a) Setze die Lösung folgender Gleichung als Δy:
$$a(\Delta y, v) + (2c\Delta y(\lambda_k + c(y_k - \psi)), \chi_{\mathcal{A}_{k+1}} v) = (f, v) - a(y, v) - ([\lambda_k + c(y_k - \psi)]^2, \chi_{\mathcal{A}_{k+1}} v)$$
   b) Update $y_{k+1} = y_k + \Delta y$
4. Setze
$$\lambda_{k+1} = \begin{cases} 0 & , \text{falls } x \in \Omega_1, \\ \lambda_k + c(y_{k+1} - \psi) & , \text{falls } x \in \Omega_2. \end{cases}$$

5. Stop oder setze $k = k + 1$ und gehe zu (2)

Der Schritt 3 kann dabei mehrmals hinter einander ausgeführt werden, bis sich y_{k+1} kaum noch verändert. In der Implementierung wird dieser Schritt zweimal durchlaufen, bevor eine ausreichend genaue Konvergenz erreicht ist.

Diesen Algorithmus ist in der Funktion `solve_state_vi(...)` implementiert:

```
def solve_state_vi(meshData, fValues, psi_values, c, tol):
```

Als Input benötigt die Funktion ein `MeshData` Objekt, das am Anfang dieses Kapitels eingeführt wurde und alle Daten zum Gitter enthält. Weiter müssen der Funktion die Funktionswerte der gebietsweise konstanten rechten Seite f unserer Zustandsgleichung übergeben werden und sind in der Variable `fValues` gespeichert. Die vorher ausgewählte Beschränkung ψ ist ebenfalls Teil des Inputs unter dem Namen `psi_values`. Dann fehlen noch die Konstante `c` für die regularisierte Formulierung und die Toleranz `tol`, durch die gewünschte Genauigkeit der Lösung festgelegt wird.

Als erstes wird der Funktionenraum der finiten Elemente bestimmt:

```
  V = FunctionSpace(meshData.mesh, "P", 1)
```

Dadurch erzeugen wir den Funktionenraum der finiten Elemente, basierend auf unserem Gitter. Das zweite Argument gibt dabei die Art des Elementes an, hier `'P'`. Dieser Buchstabe steht für die standard Lagrange-Familie von Elementen. Mit dem dritten Argument `1` geben wir den Grad der finiten Elemente an. In unserem Fall das lineare standard P_1 Lagrange Element im zwei dimensionalen Fall, welches der Dimension unseres Gitters entspricht. Hierbei handelt es sich um ein Dreieck mit Knoten an den drei Ecken. Es wird laut [6] auch als *linear triangle* bezeichnet. Die berechnete Lösung ist dabei stetig über die Elemente hinweg und variiert linear im inneren des Elementes. Eine gute Übersicht über eine Vielzahl von Elementen bietet die sogenannte *Periodic Table of the Finite Elements*[4].

Dann wird die Beschränkung ψ als ein `Function` Objekt der FEniCS Bibliothek auf dem Funktionenraum V erzeugt und anschließend wird der konstante Werte auf das gewählte Gitter interpoliert:

```
  psi = Function(V)
  psi.interpolate(psi_values)
```

Anschließend wird, wie in Schritt 1 beschrieben, λ auf Null gesetzt, im Code unter dem Namen `lmbda_c`.

```
  zero_function = Expression("0.0", degree=1)
  lmbda_c = Function(V)
  lmbda_c.interpolate(zero_function)
```

Ebenfalls wird eine weiteres `Function` Objekt als Nullfunktion über dem Gitter definiert. Hierbei handelt es sich um die Variable `abort`, dessen L_2 Norm später das Abbruchkriterium definiert.

```
  abort = Function(V)
  abort.interpolate(zero_function)
```

Auf dem äußeren Rand $\partial\Omega$ des Quadrates setzen wir die Funktion y per Dirichlet Randbedingung auf 0. In `meshData.boundaries` gibt es für jeden Knoten unseres Gitters auf dem Rand eine Zahl, die diesem zuordnet zu welchen Rand er gehört. Die vier äußeren Ränder werden mit den Ziffern 1 bis 4 durchnummeriert. In FEniCS wird so mit dem Befehl `DirichletBV(V, y_out, meshData.boundaries, 1)` allen Knoten auf dem zum Beispiel linken Rand der Wert aus der Variable `y_out` zugewiesen, basierend auf dem gewählten Raum der finiten Elementen `V`.

```
  y_out = Constant(0.0)
  bcs = [
```

[4] `https://www.femtable.org/` [gesehen am 01.05.2017]

```
        DirichletBC(V, y_out, meshData.boundaries, 1),
        DirichletBC(V, y_out, meshData.boundaries, 2),
        DirichletBC(V, y_out, meshData.boundaries, 3),
        DirichletBC(V, y_out, meshData.boundaries, 4)]
```

FEniCS ermöglicht es uns sehr einfach die schwache Formulierung zu implementieren, da die Schreibweise sehr der mathematischen Weise ähnelt. Dadurch benötigen wir auch hier ein Maß für das Integral:

```
dx = Measure('dx', domain=meshData.mesh, subdomain_data=meshData.subdomains)
```

Da wir nur Volumenintegrale implementieren müssen definieren wir im `Measure` Objekt unsere `domain` mit dem Gitter und als `subdomain_data` übergeben wir die Informationen der Teilgebiete unseres Gitter, die in der Variable `meshData.subdomains` gespeichert werden. Durch `dx(1)` und `dx(2)` können wir jeweils über die Teilgebiete Ω_1 bzw. Ω_2 integrieren.

Bevor wir die schwache Formulierung aufstellen, setzen wir die Funktionswerte als `Constant` Objekt. Wir definieren hier für jedes Teilgebiet ein eigenes Objekt, da wir später separate Integrale über die zwei Teilgebiete bilden.

```
f1 = Constant(fValues[0])
f2 = Constant(fValues[1])
```

Für die schwache Formulierung definieren wir unsere Testfunktionen `v`. Die Variable nach der später gelöst werden soll, muss für die Aufstellung der schwachen Formulierung zuerst als `TrialFunction` deklariert werden.

```
y = TrialFunction(V)
v = TestFunction(V)
```

Dann implementieren wir die Variationsformulierung. Hier ist gut die Ähnlichkeit zu der mathematischen Schreibweise zu erkennen. Das Integral $\int_\Omega fp\,\mathrm{d}x$ ist dabei in zwei Teile zerlegt worden, einmal $\int_{\Omega_1} fp\,\mathrm{d}x$ und $\int_{\Omega_2} fp\,\mathrm{d}x$.

```
a = inner(grad(y), grad(v))*dx('everywhere')
b = f1*v*dx(1) + f2*v*dx(2) - lmbda_c*v*dx('everywhere')
```

Die Lösung ohne Beschränkung, da am Anfang $\lambda_0 = 0$ gilt, erhalten wir durch:

```
y = Function(V)
solve(a == b, y, bcs)
```

Hier ist zu beachten, dass vor der Lösung die zu lösende Variable als `Function` Objekt deklariert werden muss. Für die Aufstellung der Variationsungleichung war sie ein `TrialFunction` Objekt. Dem `solve` Befehl übergeben wir in der ersten Komponente, welches System gelöst wird und in der zweiten geben wir an, nach welcher Größe das geschehen soll. Zum Schluss werden die zuvor definierten Randbedingungen übergeben, die einzuhalten sind.

Anschließend beginnen mit der Schritte 2 bis 5 des lmPDAS Algorithmus, die nachfolgend noch stückweise genauer beschrieben werden.

```
nrm = 1.0
while nrm > tol:

  # Maximumsfunktion definieren
  max_func = project(lmbda_c+c*(y-psi))
  ind = max_func.vector()<=DOLFIN_EPS
  max_func.vector()[ind] = 0.0

  # Charakteristische Funktion definieren
  chi_delta_y = Function(V)
  chi_delta_y.interpolate(Expression("1.0", degree=1))
```

```
    chi_delta_y.vector()[ind] = 0.0

    for i in range(0,2):
      delta_y = TrialFunction(V)
      lhs = (inner(grad(delta_y), grad(v))*dx('everywhere')
             + 2*c*(lmbda_c+c*(y-psi))*delta_y*v*chi_delta_y*dx('everywhere'))
      rhs = (f1*v*dx(1) + f2*v*dx(2) - inner(grad(y), grad(v))*dx('everywhere')
             - (lmbda_c+c*(y-psi))**2*v*chi_delta_y*dx)

      # Loesung berechnen
      delta_y = Function(V)
      solve(lhs == rhs, delta_y, bcs)
      y = project(y + delta_y, V)

    # Lambda updaten und Abbruchkriterium berechnen
    lmbda_c = project(lmbda_c+c*(y-psi))
    ind = lmbda_c.vector()<=DOLFIN_EPS
    abort = project(y-psi)

    lmbda_c.vector()[ind] = 0.0
    abort.vector()[ind] = 0.0

    nrm = norm(abort, 'L2', meshData.mesh)
```

Am Anfang der Schleife wird die Maximumsfunktion $\lambda_c := \max\{0, \bar{\lambda} + c(y - \psi)\}^2$ nachgebildet und die Indizes gespeichert, an denen der Wert kleiner gleich 0 ist.

```
    max_func = project(lmbda_c+c*(y-psi))
    ind = max_func.vector()<=DOLFIN_EPS
    max_func.vector()[ind] = 0.0
```

Anschließend wird anhand der vorher gespeicherten Indizes die charakteristische Funktion der Menge $\mathcal{A}_{k+1} = \{x : (\lambda_k + c(y - \psi))(x) > 0\}$ implementiert, die später bei der Implementierung der schwachen Formulierung eine Rolle spielt.

```
    chi_delta_y = Function(V)
    chi_delta_y.interpolate(Expression("1.0", degree=1))
    chi_delta_y.vector()[ind] = 0.0
```

Dann wird der Schritt 3 aus dem lmPDAS Algorithmus zweimal durchgeführt.

```
    for i in range(0,2):
        delta_y = TrialFunction(V)
        lhs = (inner(grad(delta_y), grad(v))*dx('everywhere')
               + 2*c*(lmbda_c+c*(y-psi))*delta_y*v*chi_delta_y*dx('everywhere'))
        rhs = (f1*v*dx(1) + f2*v*dx(2) - inner(grad(y), grad(v))*dx('everywhere')
               - (lmbda_c+c*(y-psi))**2*v*chi_delta_y*dx)

        # Loesung berechnen
        delta_y = Function(V)
        solve(lhs == rhs, delta_y, bcs)
        y = project(y + delta_y, V)
```

Zum Schluss der Schleife wird, wie in Schritt 4, λ_{k+1} berechnet und das Abbruchkriterium aktualisiert.

```
    # Lambda updaten und Abbruchkriterium berechnen
    lmbda_c = project(lmbda_c+c*(y-psi))
    ind = lmbda_c.vector()<=DOLFIN_EPS
    abort = project(y-psi)

    lmbda_c.vector()[ind] = 0.0
    abort.vector()[ind] = 0.0

    nrm = norm(abort, 'L2', meshData.mesh)
```

Nach Definition in der regularisierten Formulierung wird λ_c quadriert, was durch folgende Anweisung erfolgt.

```
    lmbda_c_squared = interpolate(Expression("f*f", f = interpolate(lmbda_c, V), degree=1),V)
```

Dies wird auf eine umständliche Weise implementiert, nachdem bei einfacher Quadratur Anweisung fehlerhafte Werte aufgetreten sind. Für manche Punkte wurden negative Ergebnisse berechnet. Da in der adjungierten Gleichung $\sqrt{\lambda_c}$ benötigt wird, liefert die Funktion `solve_state_sovi` dafür die Variable `lmbda_c` zurück, die vor dem Quadrieren dem Wert der Wurzel $\sqrt{\lambda_c}$ entspricht. Zusätzliche Rückgabewerte sind die Lösung y und `lmbda_c_squared`, was λ_c entspricht.

```
    return y, lmbda_c, lmbda_c_squared
```

4.3 Adjungierte Gleichung

In diesem Abschnitt beschäftigen wir uns mit der Lösung der adjungierten Gleichung, die folgendermaßen aussieht:

$$\begin{aligned} -\Delta p + 2c\sqrt{\lambda_c}p &= -(y-z) && \text{in } \Omega \\ p &= 0 && \text{auf } \partial\Omega \\ [\![p]\!] &= 0 && \text{auf } S \\ [\![\tfrac{\partial p}{\partial \vec{n}}]\!] &= 0 && \text{auf } S. \end{aligned}$$

Die Variationsformulierung ist in (3.11) gegen durch

$$\int_\Omega \nabla p^T \nabla v \,\mathrm{d}x + \int_\Omega 2c\sqrt{\lambda_c} p v \,\mathrm{d}x = \int_\Omega -(y-z)v \,\mathrm{d}x + \int_S [\![\tfrac{\partial p}{\partial \vec{n}} v]\!] \,\mathrm{d}s$$

Hierbei handelt es sich um ein leicht abgewandeltes Poisson-Problem, dessen Lösung in [6] in der Standardversion abgehandelt wird, weshalb wir hier die einzelnen Schritte weniger ausführlich behandeln. Viele Anweisungen treten bei jeder Implementierung in gleicher oder ähnlicher Weise auf. Die Lösung der adjungierten Gleichung ist in folgender Funktion beschrieben:

```
def solve_adjoint(meshData, y, z, lmbda_c_root, c):
```

Als Eingabewerte werden die Gitterdaten, die Lösung der Zustandsgleichung y, die Zieldaten z und $\sqrt{\lambda_c}$ sowie der Parameter c benötigt. Die Werte für y und $\sqrt{\lambda_c}$ stammen aus der Lösung der regularisierten Gleichung, die vorher berechnet werden müssen.

Unsere Ansatzfunktionen wählen wir auch hier als *linear triangle*:

```
    V = FunctionSpace(meshData.mesh, "P", 1)
```

Auf allen vier Rändern werden die Randbedingungen entsprechend auf 0 gesetzt:

```
  p_out = Constant(0.0)
  bcs = [
         DirichletBC(V, p_out, meshData.boundaries, 1),
         DirichletBC(V, p_out, meshData.boundaries, 2),
         DirichletBC(V, p_out, meshData.boundaries, 3),
         DirichletBC(V, p_out, meshData.boundaries, 4)]
```

Dann implementieren wir das Variationsproblem in FEniCS:

```
dx = Measure('dx', domain=meshData.mesh, subdomain_data=meshData.subdomains)

p = TrialFunction(V)
v = TestFunction(V)

a = inner(grad(p), grad(v))*dx + 2*c*sqrt(lmbda_c)*p*v*dx
l = -y*v*dx + z*v*dx
```

Zum Schluss wird das Variationsproblem mit den vorher festgelegten Randbedingungen gelöst.

```
p = Function(V)
solve(a == l, p, bcs)
```

Als Rückgabe liefert uns diese Funktion die Lösung p der adjungierten Gleichung:

```
return p
```

4.4 Lineare Elastizitätsgleichung und Gitterverformung

Die lineare Elastizitätsgleichung wird mit der Funktion `solve_linelas(...)` gelöst.

```
def solve_linelas(meshData, p, y, lmbda_c, z, fValues, mu_elas, nu):
```

Als Eingabe benötigt diese die Gitterdaten, die zuvor berechneten Lösungen p und y sowie λ_c und die Zieldaten z. Außerdem werden die Funktionswerte f_1 und f_2 der gebietsweise konstanten Funktion f benötigt, die unter `fValues` abgespeichert sind. Als vorletzte Eingabe wird der lokal variierende Lamé-Parameter benötigt, hier mit `mu_elas` bezeichnet. Zum Schluss wird der Wert für den Faktor ν der Perimeter-Regularisierung angegeben, hier unter dem Namen `nu`.

Unsere Ansatzfunktionen wählen wir auch hier als *linear triangle*, aber mit `dim=2`, da wir ein Vektorfeld berechnen und keine skalare Größe:

```
V = VectorFunctionSpace(meshData.mesh, "P", 1, dim=2)
```

Aus dem selben Grund sind auch die Randbedingungen zweidimensional. Auf den äußeren vier Rändern wird als Dirichlet Randbedinung der Nullvektor festgelegt.

```
u_out = Constant((0.0, 0.0))
bcs = [
         DirichletBC(V, u_out, meshData.boundaries, 1),
         DirichletBC(V, u_out, meshData.boundaries, 2),
         DirichletBC(V, u_out, meshData.boundaries, 3),
         DirichletBC(V, u_out, meshData.boundaries, 4)]
```

Bei der Lösung der linearen Elastizitätsgleichung benötigen wir zwei Maße. Einmal eines für die Volumenintegrale, wie zuvor auch, und andererseits eines für das Oberflächenintegral über den inneren Rand, das wegen die Perimeterregularisierung verwendet werden muss.

```
dx = Measure('dx', domain=meshData.mesh, subdomain_data=meshData.subdomains)
dS = Measure('dS', subdomain_data=meshData.boundaries)
```

Die Funktionswerte f_1 und f_2 sowie die `TrialFunction` und `TestFunction` werden wie zuvor definiert. Es ist jedoch zu beachten, dass es sich bei `U` und `v` um keine skalaren Größen handelt, sondern Vektoren.

```
  f1 = Constant(fValues[0])
  f2 = Constant(fValues[1])

  U = TrialFunction(V)
  v = TestFunction(V)
```

Durch die Wahl von $\lambda_{\text{elas}} = 0$ erhalten wir in der linearen Elastizitätsgleichung die vereinfachte Version der Dehnungs- und Spannungstensoren ϵ und σ:

$$\epsilon = \tfrac{1}{2}\left(\nabla U + \nabla U^T\right),$$
$$\sigma = 2\,\mu_{\text{elas}}\,\epsilon.$$

Die beiden Tensoren können als Funktionen in Python definiert und später in der schwachen Formulierung aufgerufen, siehe [6]:

```
def epsilon(v):
  return sym(grad(v))  # = 0.5*(grad(v) + grad(v).T)

def sigma(v):
  return 2.0*mu*epsilon(v)
```

Wir definieren hier aber direkt `sigma_U` und `epsilon_v` als Variablen.

```
epsilon_v = sym(nabla_grad(v))
sigma_U   = 2.0*mu_elas*sym(nabla_grad(U))
```

Die linke Seite der schwachen Formulierung verwendet die zuvor definierten Variablen und wird unter dem Namen `LHS` (engl. *left hand side*) assembliert.

```
a = inner(sigma(U), epsilon(v))*dx('everywhere')
LHS = assemble(a)
```

Die rechte Seite setzt sich aus zwei Teilen zusammen. Der erste Teil ist die Formableitung ohne die Perimeterregularisierung. Wie bereits bei der Implementierung der regularisierten Gleichung wird auch hier das Integral $\int_\Omega \operatorname{div}(V)(fp)\,\mathrm{d}x$ in Integrale über die einzelnen Teilgebiete separiert, auf Grund der unterschiedlichen Funktionswerte f_1 und f_2. Der Rest der Formableitung ohne Regularisierung kann über den ganzen Raum implementiert werden.

```
Dj = (-inner(grad(y), dot(epsilon_v*2, grad(p)))*dx
      + nabla_div(v)*(1/2*(y-z)**2 + inner(grad(y), grad(p))+lmbda_c*p)*dx
      - nabla_div(v)*f1*p*dx(1) - nabla_div(v)*f2*p*dx(2))
```

Der zweite Teil der rechten Seite besteht aus der Implementierung der Formableitung der Perimeterregularisierung. Hier werden die Oberflächen Integrale über den inneren Rand Ω_{int} durch das zuvor definierte Maß `dS` berechnet, wobei die Indizes 5 und 6 den inneren Rand bezeichnen. Mit dem Zeichen `'+'` wird dabei eine Richtung festgelegt.

```
Dj_reg = nu*((nabla_div(v('+')) - inner(dot(nabla_grad(v('+')), n('+')), n('+')))*dS(5)
             +(nabla_div(v('+')) - inner(dot(nabla_grad(v('+')), n('+')), n('+')))*dS(6))
```

Nach dem Aufstellen der zwei Teile werden diese zusammenaddiert und mit Hilfe von `assemble` assembliert.

```
F_elas = assemble(Dj + Dj_reg)
```

Allen Knoten, die nicht zum inneren Rand gehören, wird der Wert 0 zugeordnet. Diese sollen nach den Überlegungen aus dem vorherigen Kapitel keinen Einfluss auf die Lösung, hier dem Deformationsvektorfeld haben.

```
F_elas[meshData.indNotIntBoundary] = 0.0
```

Um die Randbedingungen einzuhalten, werden diese auch noch auf die assemblierten Formen angewendet.

```
  for bc in bcs:
    bc.apply(LHS)
    bc.apply(F_elas)
```

Um eine Abbruchbedingung des Formtoptimierungsverfahrens festzulegen wird die L_2-Norm der assemblierten rechten Seite berechnet.

```
  nrm_f_elas = norm(F_elas, 'L2', meshData.mesh)
```

Schließlich wird das Problem gelöst und das Deformationsvektorfeld in der Variable `U` gespeichert.

```
  U = Function(V, name="deformation_vec")
  solve(LHS, U.vector(), -F_elas)
```

Als Rückgabe liefert diese Funktion das Deformationsvektorfeld und die Abbruchbedingung.

```
  return U, nrm_f_elas
```

Nach der Berechnung des Deformationsvektorfeldes können wir das Gitter verformen. Dazu wird folgender Befehl verwendet, bei dem das Gitter `MeshData.mesh` in die Richtung der Deformation `U` verschoben wird. Dazu wird jeder Knoten im Gitter in die Richtung verschoben, die das Deformationsvektorfeld an jedem Punkt vorgibt. Der Betrag des Vektorfeldes an jedem Punkt gibt dabei die Stärke der Verschiebung an.

```
ALE.move(MeshData.mesh, U)
```

Um das gesamte Verfahren zu beschleunigen ist es denkbar ein BFGS Update zu implementieren, analog zu [9].

4.4.1 Lamé-Parameter Berechnung

Die Berechnung des lokalen Lamé-Paramters μ_{elas} ist in der Funktion `calc_lame_par(...)` implementiert.

```
def calc_lame_par(meshData, mu_min_value, mu_max_value):
```

Als Eingabe benötigt diese Funktion die Gitterdaten sowie die Randbedingungen $\mu_{\min}$ und $\mu_{\max}$.

Unsere Ansatzfunktionen wählen wir wie zuvor als *linear triangle*:

```
  V = FunctionSpace(meshData.mesh, "P", 1)
```

Die Randbedingungen werden entsprechend der Input-Parameter festgelegt:

```
  mu_min = Constant(mu_min_value)
  mu_max = Constant(mu_max_value)
```

Der minimale Wert wird dabei am äußeren Rand angelegt. In unserem Fall haben die vier äußeren Ränder die Indizes $1, 2, 3$ und 4. Am inneren Rand wird der maximale Wert festgelegt. Im Falle der Kreise besteht dieser aus zwei Halbkreisen mit den Indizes 5 und 6. Die Ellipse wurde aus einem einzigen Spline erstellt, womit dieser Linie nur der Index 5 zugeordnet wird.

```
bcs = [
DirichletBC(V, mu_min, meshData.boundaries, 1),
DirichletBC(V, mu_min, meshData.boundaries, 2),
DirichletBC(V, mu_min, meshData.boundaries, 3),
DirichletBC(V, mu_min, meshData.boundaries, 4),
DirichletBC(V, mu_max, meshData.boundaries, 5),
DirichletBC(V, mu_max, meshData.boundaries, 6)]
```

Anschließend erfolgen die obligatorischen Definitionen des Maßes für die Variationsformulierung sowie die `TrialFunction` und die `TestFunction`, um das Variationsproblem aufzustellen.

```
dx = Measure('dx', domain=meshData.mesh, subdomain_data=meshData.subdomains)
mu_elas = TrialFunction(V)
v = TestFunction(V)
```

Die schwache Formulierung wird folgendermaßen implementiert, wobei die rechte Seite als Nullfunktion definiert wird.

```
f = Expression("0.0", degree=1)
a = inner(grad(mu_elas), grad(v))*dx
l = f*v*dx
```

Zum Schluss löst die Funktion `solve` das Variationsproblem und wir erhalten den lokal variierenden Lamé-Parameter μ_{elas}, der in der Variable `mu_elas` gespeichert wird.

```
mu_elas = Function(V, name="lame_par")
solve(a == l, mu_elas, bcs)
```

Die Lösung μ_{elas} ist auch die einzige Rückgabe dieser Funktion, da der zweite Lamé-Parameter λ_{elas} auf 0 gesetzt ist.

```
return mu_elas
```

4.5 SOVI-Toolbox

Der Formoptimierungsalgorithmus bei Variationsungleichungen wird unter dem Namen

`SOVI-Toolbox`

implementiert. Die Abkürzung `SOVI` der Toolbox steht für *Shape Optimization with Variational Inequalities* und sie ist gemäß Abbildung 4.2 aufgebaut. Das Verfahren aus dem vorherigen Kapitel ist in der Datei `main_sovi.py` programmiert. Sie ist die Hauptdatei, in der ebenfalls alle Parameter definiert und das vom Benutzer gewünschte Gitter und die Beschränkung ψ ausgewählt werden. Um den Algorithmus zu starten, muss die Datei mit dem Befehl

`python main_sovi.py -p [Beschraenkung] -m [Gitterkombination]`

ausgeführt werden. Zur Lösung der Zustands- und adjungierten sowie der linearen Elastizitätsgleichung werden dabei speziell geschriebene Funktionen aus der Bibliothek `sovi_bib.py` verwendet. Die Implementierung der einzelnen Lösungen wurden in diesem Kapitel abschnittsweise eingeführt. Es können auch selbst erstellte Gitter verwendet werden, solange sie aus zwei Teilgebieten bestehen und diese in Gmesh als `Physical Surface` deklariert werden, um die entsprechende Datei im `.xml`-Format bei der Konvertierung zu erhalten, die notwendig ist um den Teilgebieten einen eigenen Funktionswert zuzuweisen. Die Ränder, sowohl die inneren als auch die äußeren, müssen als `Physical Line` definiert werden, um ihnen später im Programm Randbedingungen zuordnen zu können. Das genauere Vorgehen bei der Gittererzeugung wurde bereits am Anfang dieses Kapitels näher beschrieben.

Schließlich betrachten wir die Struktur, in der die Ergebnisse des Verfahrens, gespeichert werden. Dafür wird in der `SOVI Toolbox` ein Order `Output` angelegt, falls dieser noch nicht vorhanden ist. Für jeden Lauf des Programmes erstellt das Skript einen Unterorder, dessen Namen sich aus dem aktuellen Datum und der Zeit zusammensetzt, um immer einen eindeutigen Speicherort zu erhalten, der wie in Abbildung 4.3 strukturiert ist. In den unterschiedlichen Ordnern befinden sich `.pvd`-Dateien, die die berechneten Ergebnisse in jeder Iteration speichern. Diese werden im `.pvd`-Format gespeichert, um sie mit dem open-source Programm ParaView zu visualisieren. Dort ist es möglich die Berechnungen in 2D

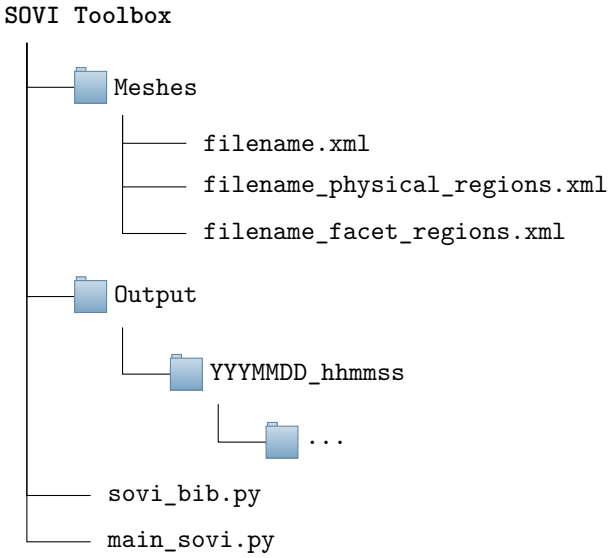


Abbildung 4.2: Aufbau der `SOVI-Toolbox`: In `main_sovi.py` wird das gewünschte Gitter und die Beschränkung ψ und alle weiteren Parameter festgelegt. Das ist auch die Hauptdatei, die mit Python ausgeführt werden muss, um das Verfahren zu starten. Dabei wird auf Funktionen und Klassen aus der Bibliothek `sovi_bib.py`. Dort sind die Verfahren zur Lösung der Zustands-, adjungierten und linearen Elastizitätsgleichung implementiert. Die `.xml`-Dateien der Gitter sind im Order `Meshes` abgelegt und werden von dort geladen. Im `Output` Ordner werden die Ergebnisse gespeichert, die in jeder Iteration entstehen sowie den berechneten lokal variierenden Lamé-Parameter und Informationen zu den Zieldaten in einem separaten Order (siehe Abbildung 4.3).

oder 3D zu betrachten und sogar den zeitlichen Verlauf animieren zu lassen. Tutorials zu diesem Programm finden sich im Internet auf der Webseite von ParaView[5]. Im Unterordner `Deformation` befindet sich das Deformationsvektorfeld als Ergebnis der linearen Elastizitätsgleichung. Dabei werden, wie im vorherigen Kapitel erläutert, lokal variierende Lamé-Parameter verwendet. Um ein besseres Bild von ihnen zu bekommen, wird die Lösung der partiellen Differentialgleichung μ_{elas} ebenfalls exportiert, hier in den Ordner `LameParameter`. Die Dateien zu jeder Iteration in diesem Ordner sind nur Projektionen der Lösung auf das aktuelle Gitter. Die einzelnen Knoten verschieben sich, der berechnete Lamé-Parameter bleibt jedoch über die Zeit konstant. Im Ordner `Mesh` ist das Gitter in jedem Iterationsschritt gespeichert, wodurch wir die Verformung im zeitlichen Verlauf betrachten können und vor allem dazu in der Lage sind die Ergebnisse mit dem Gitter der gewünschten Zieldaten vergleichen können. Die Lösungen der Zustandsgleichungen werden aus dem selben Grund in einem eigenen Order `StateSolution` gespeichert, in dem auch das Zielgitter zu finden ist. Außerdem befindet sich im Ausgabeordner eine `.csv`-Datei, die den Wert der Zielfunktion ausgibt, die L_2 Norm des Betrages vom Deformationsvektorfeldes sowie die L_2 Norm der assemblierten Formableitung in jeder Iteration enthält. Außerdem sind in dieser Datei Angaben zu den gewählten Gitterdateien und eine Auflistung der wichtigen Parameter gespeichert. Die Dateien für die Zieldaten sind in einem separaten Ordner gespeichert, dessen Struktur in Abbildung 4.4 dargestellt ist. Er enthält ebenfalls `.pdv`-Dateien zum Gitter, die Lösung der regularisierten Zustandsgleichung darauf und dem dazugehörigen λ. In `lambda.pvd` sind von Null verschiedene Werte enthalten, falls die Beschränkung durch ψ aktiv ist.

Die verschiedenen enthaltenen Gitter sind in Abbildung 5.1 zu finden, wobei die Kombinationen, die beim Starten des Skripts gewählt werden können aus Tabelle 5.1 zu entnehmen sind. Die Namen der Git-

[5] `http://www.paraview.org/tutorials/` [gesehen am 01.05.2017]

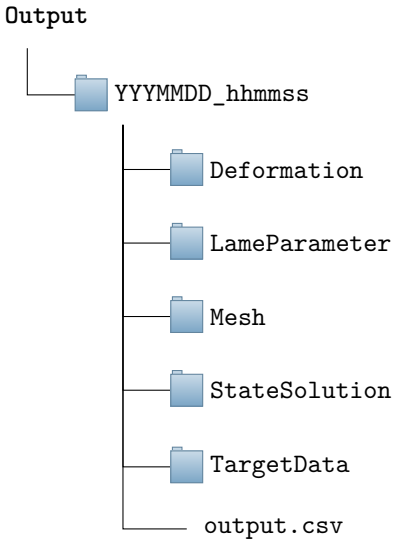


Abbildung 4.3: Im `Output` Ordner werden die in jeder Iteration berechneten Ergebnisse gespeichert. Dazu zählen das Deformationsvektorfeld als Lösung der linearen Elastizitätsgleichung, das durch diese Verschiebung entstandene Gitter und die Lösungen der Zustands- und adjungierten Gleichung. Diese Daten werden als `.pdv`-Datei in separaten Ordnern gespeichert. Außerdem werden die lokal variierenden Lamé-Parameter des zu deformierenden Ausgangsgitters gespeichert, ebenfalls als `.pdv`-Datei. Die Daten zu der Ziellösung finden sich zusammengefasst in einem eigenen Ordner (siehe dazu Abbildung 4.4). Zusätzlich wird in jeder Iteration der L_2 Fehler in einer `.csv`-Datei gespeichert, die ebenfalls noch Informationen zu den verwendeten Parametern und den ausgewählten Gittern enthält.

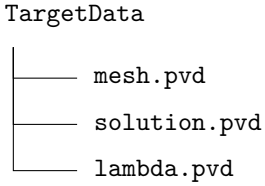


Abbildung 4.4: Im Order `TargetData` sind verschiedene *.pvd*-Dateien bezüglich der gewählten Ziellösung gespeichert. Dazu gehören das Gitter, sowie die Lösung der Zustandsgleichung mit dazugehörigem λ, falls eine Variationsungleichung eingehalten werden muss.

terdateien für die gewählte Kombination werden in den Variablen `startMesh_file` und `targetMesh_file` gespeichert.

```
(startMesh_file, targetMesh_file) = mesh_combination_list[select_combination - 1]
```

Zusätzlich zur Gitterkombination benötigt das Programm einen Wert für die Beschränkung ψ. Hier wird diese als konstant über dem gesamten Gebiet festgelegt. Schräge oder andere Formen der Beschränkung sind ebenfalls möglich, jedoch zur Vereinfachung nicht implementiert. Der vom Benutzer eingegebene Wert wird überprüft und anschließend für die weitere Verwendung als `Constant` Objekt deklariert.

```
psi_values = Constant(psi_parsed)
```

Die in jedem Teilgebiet konstante Funktion f kann durch die Variablen `f1` und `f2` bestimmt werden, die anschließend als Liste gespeichert werden.

```
f1 = -1.0
f2 = 10.0
f_values = [f1, f2]
```

Für das Verfahren zur Lösung der regularisierten Formulierung, das bereits implementiert wird, legen wir den Wert für die Konstante c und die Toleranz des Abbruchkriteriums fest.

```
c = 5.0
tol_ssn = 1.e-3
```

Für das Formoptimierungsverfahren wird eine eigene Toleranz definiert.

```
tol_shopt = 1.e-3
```

Der Faktor ν für die Perimeter-Regularisierung im Zielfunktional wird, wie in [11], auf den Wert 0,01 gesetzt.

```
nu = 0.01
```

Vor Beginn des eigentlichen Verfahrens benötigen wir dazu Zieldaten auf einem vorher festgelegtem Gitter, das wir nachher durch Verformungen eines anderen Ausgangsgitters erhalten wollen. Zur Überprüfung der Ergebnisse berechnen wir unsere Messdaten als Lösung des regularisierten Problems abhängig von der Wahl der gewünschten Kombination. Dazu wird zuerst das Gitter geladen

```
targetMeshData = sovi.load_mesh(targetMesh_file)
```

und anschließend das regularisierte Problem gelöst

```
y_z, lmbda_c, lmbda_c_squared = sovi.solve_state_vi(targetMeshData, f_values, psi_values, c,
                                                    tol_ssn)
```

Die Messdaten werden noch durch eine Normalverteilung mit den Parametern $\mu = 0$ und $\sigma = \frac{5}{100} \max_{\Omega}(|y_z|)$ gestört. Die Abweichung ist damit abhängig vom 5%-tigen, betragsmäßigen Maximalwert der Lösung im gesamten Gebiet. Das gewährleistet eine angemessene Störung je nach berechneter Lösung für ein Gitter und Beschränkung durch ψ. Die Zufallszahlen werden mit dem Paket `numpy`, das unter dem Namen `np` hier importiert wird, generiert.

```
mu = Constant(0.0)

y_z_min = y_z.vector().array().min()
y_z_max = y_z.vector().array().max()
sigma = Constant(max(abs(y_z_min), y_z_max)*5/100)

y_z.vector()[:] = y_z.vector()+np.random.normal(mu, sigma, y_z.vector().size())
```

Eine Iteration des Formoptimierungsverfahrens lässt sich wie folgt formulieren:

1. Interpolation der Zieldaten z und des Lamé-Parameters μ_{elas} auf das aktuelle Gitter
2. Lösen der Zustandsgleichung
3. Lösen der Adjungiertengleichung
4. Lösen der linearen Elastizitätsgleichung
5. Verschieben des aktuellen Gitters anhand der Lösung aus Schritt 4

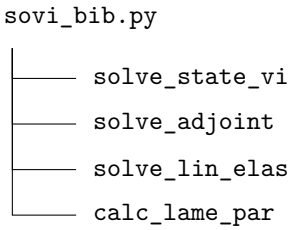


Abbildung 4.5: Übersicht über die Funktionen in der Bibliothek `sovi_bib.py`. In der Funktion `solve_state_vi.py` befindet sich das Verfahren zur Lösung der Zustandsgleichung mit potentieller Beschränkung durch eine Variationsungleichung. `solve_adjoint` löst das Problem der adjungierten Gleichung. Um das Deformationsvektorfeld zu Erhalten löst `solve_lin_elas` das lineare Elastizitätsproblem mit den lokal variierenden Lamé-Parametern, die mit Hilfe der Funktion `calc_lame_par` berechnet werden.

Die auftretenden partiellen Differentialgleichungssyteme in den Schritten $2-4$ werden mit Hilfe der dafür geschriebenen Funktionen aus der Bibliothek `sovi_bib.py`, siehe Abbildung 4.5, gelöst.

Vor der Schleife werden das Ausgangsgitter geladen und die lokal variierenden Lamé-Parameter berechnet, wobei die Funktion `calc_lame_par` im Abschnitt der linearen Elastizitätsgleichung näher beschrieben wurde.

```
MeshData = sovi.load_mesh(startMesh_file)

lmbda_elas, mu_elas = sovi.calc_lame_par(MeshData, mu_min, mu_max)
```

Die einzelnen Schritte des Verfahrens werden folgendermaßen implementiert.

```
nrm_f_elas = 1.0
while nrm_y_delta > tol_shopt:
V = FunctionSpace(MeshData.mesh, "P", 1)

# Schritt 1
z = project(y_z, V)
mu_elas_projected = project(mu_elas, V)

# Schritt 2
y, lmbda_c, lmbda_c_squared = sovi.solve_state_vi(MeshData, f_values, psi_values, c, tol_ssn)

# Schritt 3
p = sovi.solve_adjoint(MeshData, y, z, lmbda_c, c)

# Schritt 4
U, nrm_f_elas = sovi.solve_linelas(MeshData, p, y, lmbda_c_squared, z, f_values,
                                   mu_elas_projected, nu)

# Schritt 5
ALE.move(MeshData.mesh, U)
```

Mit der Anweisung

```
V = FunctionSpace(MeshData.mesh, "P", 1)
```

erzeugen wir den Funktionenraum der finiten Elemente, basierend auf unserem Gitter. Diesen Befehl haben wir bereits in den vorherigen Abschnitten verwendet. Um die Funktionen und Objekte, die von der FEniCS Bibliothek bereitgestellt werden, nutzen zu können, müssen wir diese zu Beginn des Skriptes importieren.

```
from fenics import *
```

Die anderen angesprochenen Funktionen, die zum Beispiel in Abbildung 4.5 erwähnt werden oder schon im Verlauf des Abschnittes aufgetaucht sind, können durch Importieren der `sovi_bib.py` Bibliothek genutzt werden, hier unter dem Namen `sovi` ab.

```
1 import sovi_bib as sovi
```

Weitere Python Module, die zum Beispiel zum Erstellen des Ausgabe Ordners oder zum Einlesen der Parameter benötigt werden, sind `argparse`, `copy`, `datetime`, `errno`, `os` und `time`.

Die Ergebnisse des Verfahrens sind im nächsten Kapitel dargestellt. Drei der fünf Kombinationsmöglichkeiten, die Ausgangs- bzw. Zielgitter betreffen, werden gezielt betrachtet.

Kapitel 5

Auswertung der Ergebnisse

In diesem Kapitel werden die Ergebnisse des Verfahrens in drei Fällen genauer analysiert. Im ersten Fall wird von einem kleinen Kreis ausgegangen, während die Messdaten auf einem Gitter mit einem größeren inneren Kreis erzeugt wurden. Es wird also in gewisser Weise ein kleiner Kreis zu einem größeren deformiert. Hierbei handelt es sich um einen symmetrischen Fall um den Mittelpunkt $(0{,}5\ 0{,}5)^T$ des Gebietes Ω. Um einen nicht symmetrischen Fall abzudecken werden die Messdaten auch auf einem Gitter mit einem leicht vom Mittelpunkt verschobenen Kreises erzeugt. Im einem dritten Fall werden die Messdaten wieder auf einem Gitter mit zentriertem Kreis berechnet, jedoch wird diesmal von einem ellipsenförmigen Gitter ausgegangen. Die verschiedenen Gitter sind in Abbildung 5.1 dargestellt, während die Kombinationen in Tabelle 5.1 zu finden sind.

In den einzelnen Abschnitten wird das Konvergenzverhalten und die berechneten Deformationen visualisiert. Dabei werden die Verformungen im zeitlichen Verlauf betrachtet. Es wird aber auch das deformierte Gitter dargestellt. Auf eine Analyse der Anzahl Iterationen und die dafür benötigte Rechenzeit für unterschiedliche Beschränkungen wird jedoch verzichtet, da durch die Störung der Messdaten durch normalverteilte Werte die Rechenzeit sogar bei komplett gleichen Parametern schwanken kann. Je nach Situation werden mehr oder weniger Iterationen benötigt, was auch die benötigte Zeit beeinflusst. Aber auch die Zeit pro Iteration kann je nach Gegebenheit variieren, da durch eine anders verfälschte Ziellösung unterschiedliche Zwischenschritte berechnet werden. Aus diesem Grund beschränken wir uns hier auf die Analyse eines beliebigen Durchlaufes des Verfahrens.

In allen Testfällen werden dabei die Funktionswerte $f_1 = -10$ und $f_2 = 100$ verwendet, wodurch die Funktion f des Zustandsproblems folgendermaßen aussieht

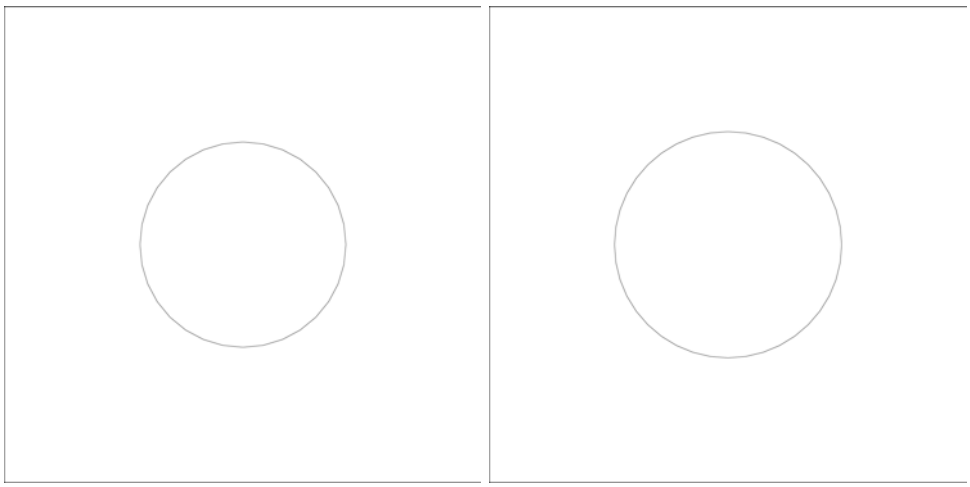
$$f(x) = \begin{cases} \text{-10} & \text{, falls } x \in \Omega_1, \\ 100 & \text{, falls } x \in \Omega_2. \end{cases}$$

Dadurch nimmt die Lösung im inneren Gebiet Ω_2 große Werte an, die durch die jeweilige Wahl von ψ beschränkt werden. Durch $f_1 = -10$ können wir das unterschiedliche Verhalten der Lösung bei verschiedenen Beschränkungen betrachten. So werden, wie in Abbildung 3.1 zu sehen, bei kleinen Beschränkung sogar negative Werte im Gebiet Ω_1 angenommen.

Die anderen Parameter sind folgendermaßen gewählt:

$$\begin{aligned} \mu_{\min} &= 1, \\ \mu_{\max} &= 20, \\ c &= 5, \\ tol_{\text{ssn}} &= 1e^{-3} && \text{(Toleranz bei der Lösung der Variationsungleichung)}, \\ \nu &= 0{,}01 && \text{(Faktor der Perimeter-Regularisierung)}, \\ tol_{\text{shopt}} &= 1e^{-3} && \text{(Toleranz bei der Lösung der Formoptimierung)}. \end{aligned}$$

Wie im vorherigen Kapitel bereits vorgestellt, sind Gitterdaten zu den nachfolgenden Formen vorhanden, die den inneren Rand beschreiben. Innerhalb dieser Form befindet sich das Gebiet Ω_2, während der Teil außerhalb davon mit Ω_1 bezeichnet wird. Der quadratische äußere Rand beschränkt das gesamte Gebiet Ω auf $(0,1)^2 \in \mathbb{R}^2$.



(a) kleiner zentrierter Kreis K_k mit Mittelpunkt $(0{,}5\ 0{,}5)^T$ und Radius 0,2

(b) großer zentrierter Kreis K_g mit Mittelpunkt $(0{,}5\ 0{,}5)^T$ und Radius 0,25

(c) verschobener Kreis K_v mit Mittelpunkt $(0{,}45\ 0{,}45)^T$ und Radius 0,25

(d) zentrierte Ellipse E mit Mittelpunkt $(0{,}5\ 0{,}5)^T$

Abbildung 5.1: Die vier verschiedenen Formen, die den inneren Rand Γ_{int} beschreiben und deren Gitterdaten in der `SOVI-Toolbox` vorhanden sind.

Die drei Testfälle, die in den nächsten Abschnitten genauer betrachtet werden, sind die Kombinationen 1, 3 und 4 entsprechend der Tabelle 5.1. Die Bezeichnungen der Gitter sind dabei aus Abbildung 5.1 zu entnehmen.

Die Berechnungen wurden dabei unter den Hard- und Softwarespezifikationen gemäß Tabelle 5.2 durchgeführt. Dabei lief das Programm unter einem Linux System in einer virtuellen Maschine auf einem Windows Rechner.

In Abbildung 5.3 ist der Verlauf der Deformationen zu sehen. In dieser Grafik wurde der innere

Kombinationsnummer	Ausgangsgitter	Zielgitter
1	K_k	K_g
2	K_g	K_k
3	E	K_k
4	K_k	K_v
5	K_v	K_k

Tabelle 5.1: Kombinationsübersicht unter den vier vorhandenen Gittern. Dabei wird in dieser Tabelle die Form des inneren Gebietes Ω_2 beschrieben, das sich immer im gesamtem Gebiet $\Omega = (0,1)^2$ befindet.

Desktop-PC: Microsoft Windows	
CPU	Intel Core i5-3570K
Mainboard	ASRock Z77 Pro 4
RAM	16GB
Grafikkarte	AMD Radeon HD7800
Betriebssystem	Windows 7 Ultimate
Virtuelle Maschine: Oracle VM VirtualBox	
zugewiesene Prozessoren	3
RAM	4GB
Grafikspeicher	100MB
Betriebssystem	Ubuntu 16.04.1 LTS 64-bit

Tabelle 5.2: Hard- und Software-Spezifikationen.

Rand $\Omega_{\texttt{int}}$ in verschiedenen Iterationen übereinander gelegt. Die roten Markierungen kennzeichnen den Anfangs- und Endzustand. Da die Messdaten gestört sind und somit keine mögliche Lösung einer Differentialgleichung sind, endet der Algorithmus auch nicht mit dem jeweiligen Kreis, auf dem die Messdaten ursprünglich erzeugt wurden. Im ersten Fall starten wir mit dem kleineren Kreis dessen Radius Iteration für Iteration vergrößert wird bis in die Nähe des inneren Randes kommt, auf dem die Messdaten generiert wurden. Im Fall der zu verformenden Ellipse wird diese erst vergrößert und anschließend in einen kreisförmigen Zustand gebracht. Dazu werden die linke und rechte Seite gestaucht und der obere und untere Rand entsprechend auseinander gezogen. Es ist also nicht nur möglich Kreise in andere Kreise zu deformieren, sondern auch Ellipsen. Im dritten Fall ist zu sehen, dass ein Kreis zusätzlich zur Vergrößerung auch verschobenen werden kann.

Wie die Verformungen sich auf das Gitter auswirken ist in Abbildung 5.4 zu sehen. In dieser Abbildung ist ebenfalls zu erkennen, dass bei den Kreisen ein gröberes Gitter gewählt wurde, während bei der Ellipse ein feineres Gitter generiert wurde. Der Algorithmus liefert sogar für die gröberen Gitter zufriedenstellende Ergebnisse. Links und rechts in dieser Abbildung ist jeweils das Gitter am Anfang bzw. Ende der Iteration abgebildet. In der Mitte ist eine Iteration dazwischen visualisiert. Bei nicht allzu großen Deformationen vergrößern sich die finiten Elemente auch nicht sonderlich. Im Fall der Ellipse ist jedoch zu erkennen, dass die Elemente im inneren Gebiet sehr stark vergrößert wurden durch die Verformungen im Vergleich zum Ausgangszustand. Um das zu vermeiden gibt es verschiedene Lösungsansätze, deren genauere Ausführung hier nicht mehr behandelt werden können. Zum einen ist es möglich das Gitter noch weiter zu verfeinern, was jedoch an vielen Stellen nicht notwendig sein wird. Eine Verfeinerung um den inneren Rand herum wäre eine gute Alternative dazu. Zum anderen wäre es aber auch vorstellbar ein relativ grobes Gitter zu verwenden und nach ein paar Iterationen das komplette Gitter neu zu generieren, basierend auf dem verschobenen inneren Rand. In dieser Abbildung ist auch gut zu erkennen, dass das Verfahren am Anfang die größten Deformationen berechnet, da der Zwischenstand eine relativ frühe Iteration darstellt. Es wird schon früh der Bereich erreicht, auf dem die Messdaten

erzeugt wurden. In Abbildung 5.2 ist dies auch gut zu erkennen. Am Anfang sinkt das zu minimierende Zielfunktional sehr stark und flacht nach ungefähr 25 Iterationen im ersten Fall schon ab. Danach bleibt die Kurve auf einem ähnlichen Niveau. Auch die L_2 Norm der assemblierte rechte Seite der linearen Elastizitätsgleichung f_{elas} zeigt dieses Verhalten. Sie beeinflusst den Betrag der Deformation, der am Anfang größer ist und immer weiter abnimmt.

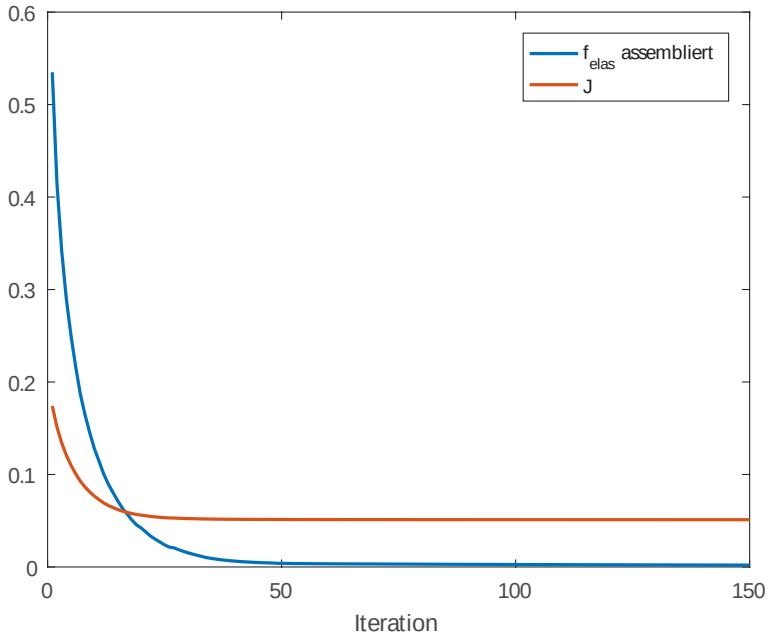
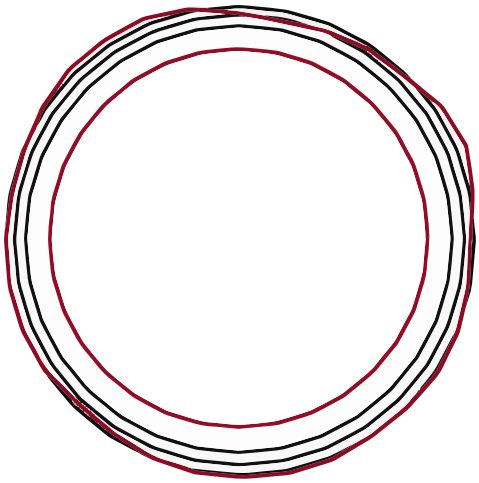
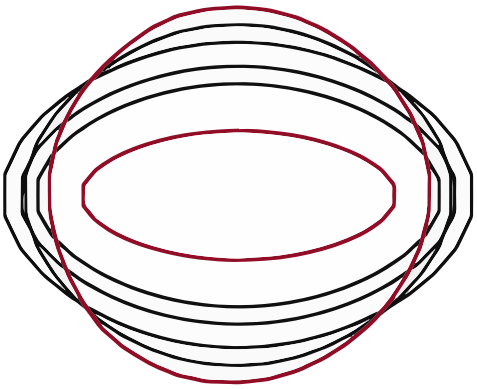


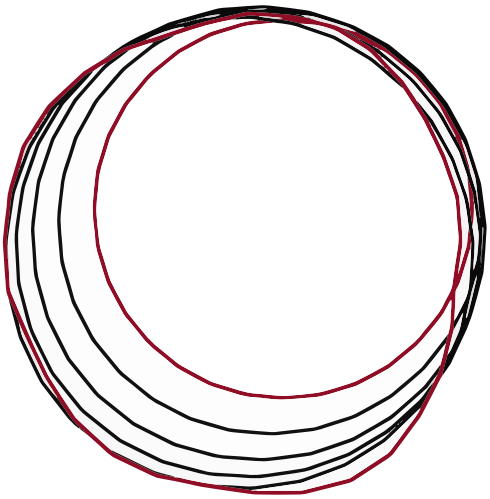
Abbildung 5.2: Der Verlauf des Zielfunktionals J ist in orange dargestellt. Die assemblierte rechte Seite der Elastizitätsgleichung f_{elas} als Gradient und Abbruchkriterium in blau. Das sind die Ergebnisse vom ersten Fall, die anderen verlaufen ähnlich. Die Kurve des Gradienten fällt am Anfang sehr stark und flacht dann hier ab der 50-ten Iteration ab. Ein ähnliches Verhalten zeigt das Zielfunktional, das jedoch schon früher abflacht und sich anschließend kaum noch verändert.



(a) Kombinationsnummer 1: Ein kleiner zentriert Kreis wird vergrößert.

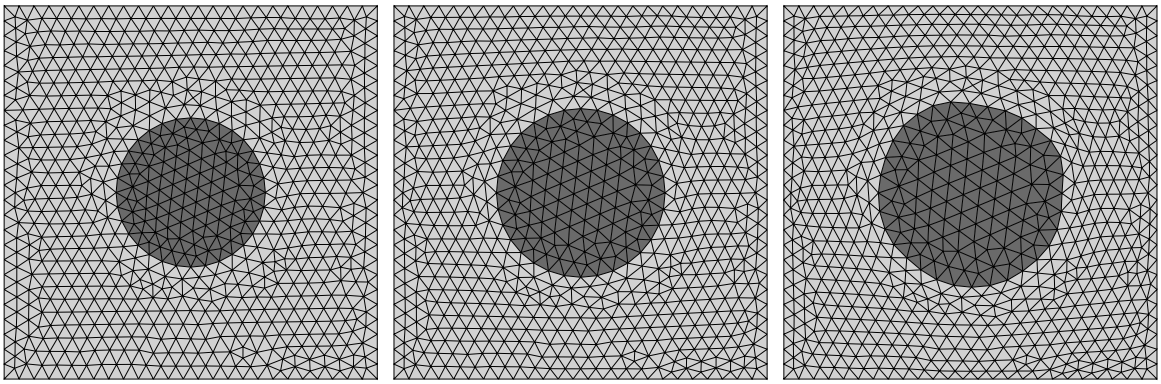


(b) Kombinationsnummer 3: Eine Ellipse wird kreisförmig.

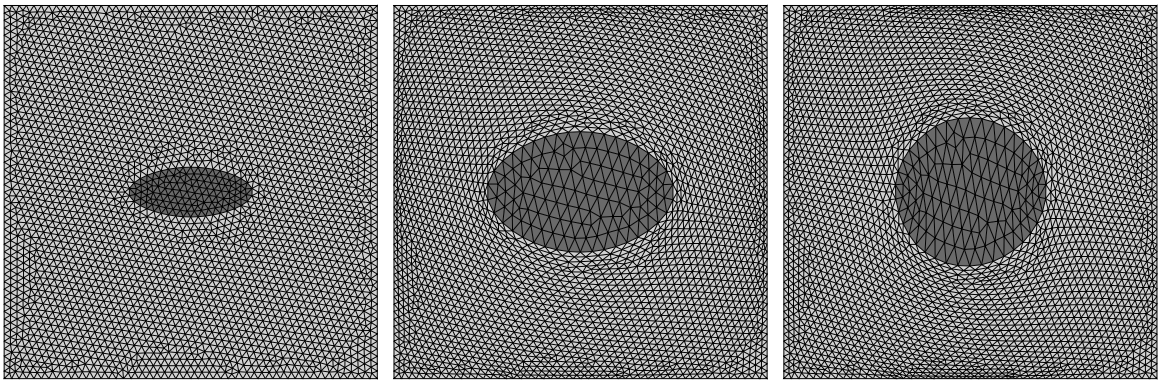


(c) Kombinationsnummer 4: Ein Kreis wird vergrößert und verschoben.

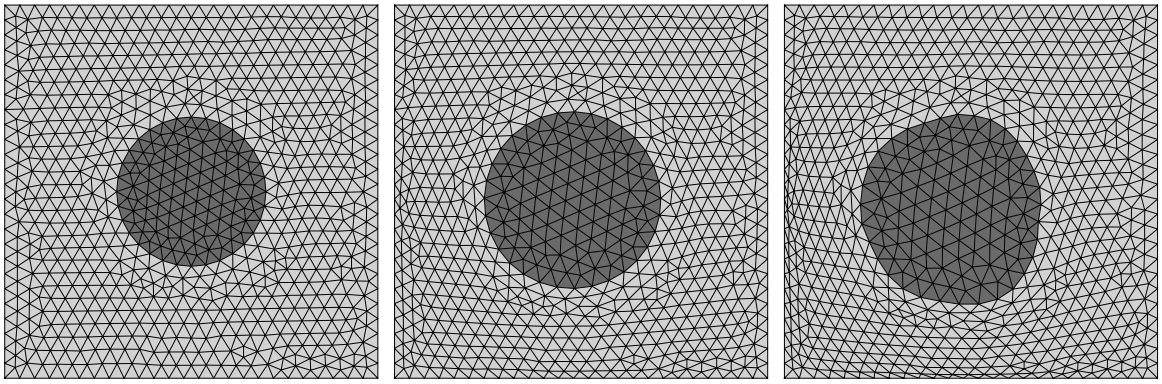
Abbildung 5.3: Verlauf der Verformung in drei Fällen. Oben wird ein kleiner Kreis vergrößert. In der Mitte ist die Verformung einer Ellipse zu sehen, während unten ein Kreis verschoben wird. Ausgangs- und Endform sind rot dargestellt, der iterative Verlauf schwarz.



(a) Kombination 1: Iteration 0 (b) Kombination 1: Iteration 2 (c) Kombination 1: Iteration 461



(d) Kombination 3: Iteration 0 (e) Kombination 3: Iteration 14 (f) Kombination 3: Iteration 396



(g) Kombination 4: Iteration 0 (h) Kombination 4: Iteration 4 (i) Kombination 4: Iteration 523

Abbildung 5.4: Ausgangs- und Endgitter mit einem Zwischenschritt. Dunkel ist dabei das Gebiet Ω_2 und hell Ω_1 eingefärbt. Die größten Verformungen finden dabei am Anfang des Verfahrens statt.

Kapitel 6

Zusammenfassung

Die Berechnung von partiellen Differentialgleichungen spielt eine große Rolle in der modernen numerischen Mathematik. Viele Probleme aus Natur und Technik können durch sie beschrieben und somit durch Computersimulationen analysiert und auch visualisiert werden. Durch den immerwährenden Fortschritt der Computertechnik sind immer genauere oder komplexere Berechnungen möglich.

In dieser Arbeit wurde erfolgreich ein Verfahren programmiert um ein Formoptimierungsproblem zu lösen, das eine Variationsungleichung beinhaltet. Das zu minimierende Zielfunktional besteht dabei aus zwei Teilen. Zum einen gibt es den kleinste Quadrate Abstand zu Messdaten an. Der zweite Teil ist eine sogenannte Perimeter-Regularisierung, um die Wohlgestelltheit zu gewährleisten. In dieser Arbeit wurde sie als Länge des inneren Randes gewählt. Das gesamte Gebiet ist in zwei Teilgebiete unterteilt. Einem inneren und äußeren, auf denen sich die Lösung der Zustandsgleichung unterschiedlich verhalten kann. Um die Implementierung jedoch zu erleichtern wurde erst einen regularisierte Formulierung der Zustandsgleichung angegeben. Anschließend betrachteten wir die dazugehörige adjungierte Gleichung und Lagrangefunktion, um die Formableitung anzugeben. Sie besteht wie das Zielfunktional ebenfalls aus zwei Teilen, wobei es sich bei einem um den Regularisierungsteil handelt. Die gesamte Formableitung wurde als rechte Seite der linearen Elastizitätsgleichung implementiert, die ein Deformationsvektorfeld berechnet, das für die Verschiebung des Gitters verantwortlich ist. Dieses Vektorfeld gibt für jeden Punkt des Gitters an in welche Richtung und um welchen Betrag er sich zu verschieben hat. Es ist ebenfalls möglich die in der Elastizitätsgleichung auftretenden Lamé-Parameter nicht über dem gesamten Gebiet konstant zu wählen, sondern sie lokal über dem Gebiet variieren zu lassen als Lösung einer weiteren partiellen Differentialgleichung.

Das Verfahren wurde dabei in der Programmiersprache Python programmiert unter der Verwendung des open-source Projektes FEniCS. Dieses hat es ermöglicht die schwache Formulierung für den Finiten-Elemente Ansatz einfach zu implementieren, da sie sehr der mathematischen Schreibweise ähnelt. Die einzelnen Probleme sowie der Formoptimierungsalgorithmus sind dabei im Rahmen der `SOVI-Toolbox` zusammengefasst. Darin sind vier verschiedene Gitter enthalten, die jeweils unterschiedliche Gebiete definieren. Beim Ausführen des Programms ist dabei einmal aus fünf verschiedenen Gitterkombinationen zu wählen. Außerdem ist die Beschränkung beim Start zu definieren. Diese wird gebietsweise konstant angenommen. Bei den Ergebnissen ist zu sehen, dass die größte Deformation am Anfang stattfindet und nachher nur noch kleine Verformungen berechnet werden. Einmal ist in den Ergebnissen die Vergrößerung eines Kreises zu sehen. Aber auch die Verschiebung ist in einem weiteren Fall zu sehen. Als drittes Beispiel ist die Deformation einer Ellipse zu einem Kreis dargestellt, um zu zeigen, dass auch andere Formen als Kreise verwendet werden können.

Ein Fazit dieser Arbeit ist, dass es mit Python und FEniCS möglich ist die einzelnen Probleme zu implementieren, da auf bereit vorhandene Objekte zurückgegriffen werden kann. Mögliche Weiterentwicklungen des Verfahrens könnte die Einführung eines BFGS-Updates sein, um den Algorithmus zu beschleunigen. Um größere Deformation des Gitters über eine weitere Strecke zu gewährleisten, ist es vorstellbar das Gitter am inneren Rand feiner zu wählen oder nach ein paar Iterationen das gesamte Gitter neu zu generieren mit dem Vorteil auf einem gröberen Gitter zu arbeiten.

Anhang A

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Masterarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Trier, 20. Oktober 2017

(Björn Führ)

Literaturverzeichnis

[1] M. Berggren. „A unified discrete-continuous sensitivity analysis method for shape optimization“. In: *Applied and Numerical PDEs: Scientific Computing in Simulation, Optimization and Control and its Multiphysik Applications* (2010).

[2] P. Deuflhard und M. Weise. *Numerische Mathematik 3: Adaptive Lösung partieller Differentialgleichungen*. De Gruyter, 2011.

[3] J. Hasslinger und M. A. E. Mäkinen. *Introduction to Shape Optimization: Theory, Approximation, and Computation*. SIAM Philadelhpia, 2008.

[4] M. Hintermüller und A. Laurain. „Optimal shape design subject to elliptic variational inequalities“. In: *SIAM Journal on Control and Optimization* (2011).

[5] K. Ito und K. Kunisch. „Semi-smooth newton methods for variational inequalities of the first kind“. In: *Mathematical Modelling and Numerical Analysis* (2002).

[6] H. P. Langtangen und A. Logg. *Solving PDEs in Minutes - The FEniCS Tutorial Volume 1*. Springer, 2016.

[7] V. Schulz und M. Siebenborn. „Computational comparison of surface metrics for PDE constrained shape optimization“. In: *Computational Methods in Applied Mathematics* vol. 16 (2016), Seiten 485–496.

[8] V. Schulz, M. Siebenborn und K. Welker. „Efficient PDE constrained shape optimization based on Steklov-Poincaré type metrics“. In: *SIAM Journal on Optimization* (2016).

[9] V. Schulz, M. Siebenborn und K. Welker. „Structured inverse modeling in parabolic diffusion processess“. In: *SIAM Journal on Optimization* (2015).

[10] V. Schulz, M. Siebenborn und K. Welker. „Towards a Lagrange-Newton approach for PDE constrained shape optimization“. In: *SIAM Journal on Optimization* (2014).

[11] M. Siebenborn und K. Welker. „Computational Aspects of Multigrid Methods for Optimization in Shape Spaces“. In: (2017).

[12] J. Sokolowski. „Displacement Derivatives in Shape Optimization of Thin Shells“. In: *[Research Report] RR-2995, INRIA* (1996).

[13] J. Sokolowski und J.-P. Zolesio. *Introduction to Shape Optimization*. Springer-Verlag, 1992.

[14] F. Tröltzsch. *Optimale Steuerung partieller Differentialgleichungen*. Vieweg+Teubner Verlag, 2009.

[15] K. Welker. „Efficient PDE Contrained Shape Optimization in Shape Spaces“. Dissertation. Universität Trier, 2016.