Quasi-Newton-Methoden in der Formoptimierung

Masterarbeit

zur Erlangung des akademischen Grades eines
Master of Science

eingereicht am Fachbereich IV
der Universität Trier

von

Daniel Luft

Erstgutachter: Prof. Dr. Volker Schulz
Zweitgutachter: Dr. Kathrin Welker

Trier, den 9. Juni 2018.

Universität Trier - Gebäude E
Fachbereich IV - Mathematik
Universitätsring 19
D-54296 Trier

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.

(Datum) (Daniel Luft)

Danksagung

Ich möchte mich an dieser Stelle bei all denen bedanken, die mir während der Erstellung dieser Masterarbeit zur Seite standen.

Bei Prof. Dr. Volker Schulz möchte ich mich für die unterhaltsamen und aufschlussreichen Gespräche bedanken.

Meiner Familie und Freunden, vor allem meinen Eltern und Aibi, möchte ich danken, immer mit Rat und Tat da gewesen zu sein.

Inhaltsverzeichnis

Zusammenfassung

Die Formoptimierung ist ein für die moderne Industrie immer wichtiger werdendes Teilgebiet der Mathematik. Durch Fortschreiten der technischen Realisierbarkeit und den immer mächtiger werdenden Rechenarchitekturen sind Simulation, Lösung und Umsetzung von Modellbasierten Ansätzen möglich, die vor einigen Dekaden undenkbar schienen. So wird die Formoptimierung heutzutage beispielsweise zur Optimierung von Bauteilen hinsichtlich ihrer Aerodynamik in [14], bei Problemen der Magnetostatik in [4], oder dem Design von Turbinen bei [12]. Aber auch bei weniger bekannten Problemstellungen, etwa dem optimalen Design von Hafenanlagen, wie bei [2], wo die Form unter anderem bezüglich Kostenintensität, aber auch der auftretenden signifikanten Wellenhöhe, optimiert wird. Ziel dieser Arbeit ist es, das Broyden-Fletcher–Goldfarb-Shanno-Verfahren (BFGS-Verfahren), als Beispiel eines Quasi-Newton-Verfahrens, im Kontext der Formoptimierung in einem Programm, welches künftig weiter verwendet werden kann, aufzusetzen und zu analysieren. Hierfür werden wir in Kapitel 1 und 2 die für uns relevanten theoretischen Bausteine aus der Theorie der Differentialgleichungen und der Formoptimierung legen. In Kapitel 3 bauen wir auf das vorangegangene auf, und führen in die moderne Betrachtung der Formoptimierung auf Basis von Formräumen ein. Wir zeigen, wie man Metriken auf diesem Raum definiert, und so zugehörige Gradienten erzeugen kann. Anschließend geben wir eine kurze Betrachtung von Newton- und Quasi-Newton-Verfahren in Formräumen, woraufhin wir in Kapitel 4 unsere Implementierung vorstellen werden. Die mit dieser Implementierung erzeugten Datensätze bereiten wir im letzten Kapitel 5 auf, und geben abschließend einen Ausblick.

1 Grundlagen zur Theorie der Differentialgleichungen

In diesem Abschnitt möchten wir eine kurze Einführung in die für diese Arbeit nötigen Grundlagen zur Theorie und Numerik partieller Differentialgleichungen geben. Im Folgenden werden wir uns an die Werke [21], [10] und [16] halten, und verweisen für weitere Details und Beweise auch auf diese.
Da wir in dieser Arbeit Formoptimierung bei Differentialgleichungen betreiben möchten, stellen wir hier die von uns ausgewählte Modellgleichung, später im

Kontext der Formoptimierung auch *Zustandsgleichung* (engl. *state equation*) genannt, vor. Es handelt sich dabei um das bekannte *Poisson-Problem*

$$\begin{aligned} -\Delta y &= f \text{ in } \Omega \\ y &= 0 \text{ auf } \partial\Omega, \end{aligned} \tag{1.1}$$

wobei Δ den Laplace-Operator meint, und Ω ein offenes, beschränktes, zusammenhängendes Gebiet des $\mathbb{R}^n$ ist. Funktionen $y \in C^2(\Omega)$, welche die obigen Gleichungen erfüllen, werden klassische oder auch *starke Lösungen* des Poisson-Problems genannt. Betrachtet man das Poisson-Problem für kompliziertere Gebiete Ω und Funktionen f auf Ω, so ist im Allgemeinen nicht klar, ob eine Lösung existiert und welche Regularität diese besitzt. Aus diesem Grund führen wir die bekannten Sobolev-Räume und exemplarische Elemente der schwachen Lösungstheorie des Poisson-Problems ein. Wir beginnen mit der Definition von Sobolev-Räumen nach [16].

Definition 1 (Sobolev-Räume)**.** Sei $\Omega \subseteq \mathbb{R}^n$ ein offenes, beschränktes, zusammenhängendes Gebiet, weiterhin sei $1 \leq p \leq \infty$, $m \in \mathbb{N}$. Wir definieren eine Norm durch

$$||\cdot||_{W^{m,p}(\Omega)} \colon C^\infty(\Omega) \to \mathbb{R},\ y \mapsto \sum_{|\alpha| \leq m} ||D^\alpha y(x)||_{\mathcal{L}^p},$$

genannt *m,p-Sobolev-Norm*, wobei α ein Multiindex ist. Dann heißt die Vervollständigung von $C^\infty(\Omega)$ bezüglich $||\cdot||_{W^{m,p}(\Omega)}$ *Sobolev-Raum der Ordnung m*, bezeichnet mit $W^{m,p}(\Omega)$. Die Vervollständigung von $C_0^\infty(\Omega)$ bezüglich $||\cdot||_{W^{m,p}(\Omega)}$ bezeichnen wir mit $W_0^{m,p}(\Omega)$. Im Falle $p = 2$ nennen wir diese Sobolev-Räume $H^m(\Omega)$, beziehungsweise $H_0^m(\Omega)$.

Beweise zu der in der obigen Definition getroffenen Behauptung, die Abbildung $||\cdot||_{W^{m,p}(\Omega)}$ definiere eine Norm, finden sich in den Eingang dieses Abschnittes genannten Werken. Die so definierten Sobolev-Räume enthalten per Konstruktion die C^∞- beziehungsweise C_0^∞-Funktionen als dichte Teilmenge, was als Satz von Meyers-Serrin bekannt ist. Ein alternativer Zugang zu den Sobolev-Räumen bietet die Definition über Abbildungen mit existierenden schwachen Ableitungen. In der Tat sind diese Definitionen äquivalent, weshalb wir uns im Hinblick auf die weiteren Sätze hier eine kurze Definition geben. Bei Interesse verweisen wir auf [16].

Definition 2 (schwache Ableitung)**.** Sei ein $\Omega \subseteq \mathbb{R}^n$ beschränktes, offenes, zusammenhängendes Gebiet. Sei $f \in \mathcal{L}^1(\Omega)$ und $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$ ein

Multiindex. Dann heißt eine Funktion $u \in \mathcal{L}^1(\Omega)$ *α-te schwache Ableitung* von f, falls gilt

$$\int_\Omega u(x)\varphi(x)dx = (-1)^{|\alpha|}\int_\Omega f(x)D^\alpha\varphi(x)dx \qquad \forall\varphi \in C_c^\infty(\Omega),$$

wobei $C_c^\infty(\Omega)$ der Raum aller $C^\infty(\Omega)$-Funktionen mit kompaktem Träger ist.

In Anlehnung an das Fundementallemma der Variationsrechnung verwenden wir nun, ausgehend von 1.1, die Green'schen Formeln und Testfunktionen $v \in C_0^\infty(\Omega)$, um die sogenannte *schwache Formulierung* des Poisson-Problems herzuleiten. Es sei n das äußere Einheitsnormalenvektorfeld von Ω. Dann folgt

$$\begin{aligned}
&\int_\Omega (-\Delta y(x))v(x)dx \\
&\overset{\text{Green}}{=} \int_\Omega \nabla y(x)\nabla v(x)dx + \int_{\partial\Omega} \frac{\partial y(s)}{\partial n(s)}v(s)ds \\
&\overset{v_{|\partial\Omega}=0}{=} \int_\Omega \nabla y(x)\nabla v(x)dx.
\end{aligned}$$

Somit lässt sich Problem 1.1 im schwachen Sinne wie folgt definieren, vgl. [16]:

Definition 3 (schwache Formulierung des Poisson-Problems)**.** Sei Ω ein offenes, beschränktes Gebiet des $\mathbb{R}^n$. Sei $f \in \mathcal{L}^2(\Omega)$. Dann heißt $y \in H_0^1(\Omega)$ *schwache Lösung* des Poisson-Problems, falls gilt:

$$\int_\Omega \nabla y(x)\nabla v(x)dx = \int_\Omega f(x)v(x)dx \qquad \forall v \in H_0^1(\Omega). \tag{1.2}$$

Aus theoretischer Sicht ist es wichtig zu klären, unter welchen Bedingungen an das Gebiet Ω und die Funktion f, Lösungen und Eindeutigkeit dieser garantiert sind. Das klassische Theorem, welches Auskunft hierüber gibt, ist das Lemma von Lax-Milgram, vgl. [16], welches wir hier nicht in vollster Allgemeinheit und ohne Beweis angeben.

Theorem 4 (Lemma von Lax-Milgram)**.** Seien H ein Hilbertraum, H^* der zugehörige Dualraum und $a(\cdot,\cdot)$ eine stetige, *koerzive* Bilinearform, d.h. es existiert eine Konstante $c > 0$, so dass gilt

$$|a(y,y)| \geq c||y||^2 \qquad \forall y \in V.$$

Weiterhin sei $b \in H^*$. Dann gibt es genau ein $\tilde{y} \in V$, so dass gilt

$$a(\tilde{y}, v) = b(v) \qquad \forall v \in V.$$

Betrachtet man die linke Seite von 1.2, so sieht man, dass das Integral eine Bilinearform auf dem zugehörigen Sobolevraum definiert. Die Koerzivität dieser Bilinearform ist gesichert durch eine Anwendung der sogenannten Poincaré-Ungleichung, siehe [16] Theorem 6.6. Das Lemma von Lax-Milgram sichert somit für das Poisson-Problem in schwacher Formulierung die Existenz und Eindeutigkeit der Lösung, und wir können im Laufe dieser Arbeit Wohlgestelltheit dieser Modellgleichung voraussetzen. Die Koerzivität als Bedingung an die Bilinearform $a(\cdot, \cdot)$ wird in diesem Kontext oft auch *V-Elliptizität* genannt, da die Bilinearform Koerzivität bezüglich der Funktionen $y \in V$ erfüllt. Offensichtlich ist jede starke Lösung des Problems 1.1 auch eine schwache Lösung in dem obigen Sinne. Andersherum gilt diese Aussage im Allgemeinen nicht, denn nicht alle schwach differenzierbaren Funktionen besitzen Ableitungen im herkömmlichen/ starken Sinne. Um sich Klarheit über die Regularität bestimmter Lösungen zu verschaffen führen wir hier die sogenannten Sobolev-Einbettungssätze zusammengefasst, und ohne Beweise, auf, vgl. [16], sowie [20].

Theorem 5 (Einbettungssätze für Sobolevräume)**.** Sei Ω ein offenes, beschränktes Gebiet des $\mathbb{R}^n$ mit Lipschitz-Rand $\partial\Omega$. Seien $k, l, m \in \mathbb{N}$ und $\alpha \in (0, 1]$, sowie $p, q \in [1, \infty)$. Bezeichne mit $C^{k,\alpha}(\Omega)$ den Hölderraum der Funktionen mit α-hölderstetigen m-ten Ableitungen. Dann gilt:
(i) (*Morrey*)
Falls $p < n$ und $m - \frac{n}{p} \geq k + \alpha$, so existiert eine Einbettung

$$W^{m,p}(\Omega) \hookrightarrow C^{k,\alpha}(\Omega).$$

Falls $m - \frac{n}{p} > k + \alpha$, so ist die Einbettung kompakt.
(ii) (*Rellich-Kondrachov*)
Falls gilt $k > l$ und $m - \frac{n}{p} > l - \frac{n}{q}$, so existiert eine kompakte Einbettung

$$W^{l,q}(\Omega) \hookrightarrow W^{m,p}(\Omega).$$

Diese Theoreme bieten uns die Möglichkeit, Aussagen über Regularität der Lösung einer PDE machen. Durch den Satz von Morrey wird auch klar, in welchen Fällen glatte, und somit starke Lösungen für das Poisson-Problem zu erhalten sind, d.h. schwache Lösungen auch starke Lösungen darstellen. Weiterhin möchten wir bemerken, dass die hier genannten Einbettungssätze auch

für die sogenannten Sobolev-Slobodeckij-Räume, welche eine Verallgemeinerung der Sobolevräume auf gebrochene Ordnungen $s \in (0, \infty]$ sind, im selben Wortlaut gelten. Diese werden wir im Laufe dieser Arbeit bei der Einführung in L-BFGS-Methoden für die Formoptimierung benötigen, weshalb wir diese im Folgenden definieren, vgl [20].

Definition 6 (Sobolev-Slobodeckij-Räume)**.** Sei $\Omega \subseteq \mathbb{R}^n$, $m \in \mathbb{N}$, sowie $p \in [1, \infty)$ und $\sigma \in (0, 1)$. Wir definieren für messbare Funktionen $u : \Omega \to \mathbb{R}$ die sogenannte *Slobodeckij-Halbnorm* durch

$$|u|_{\sigma,p} := \left(\int_\Omega \int_\Omega \frac{|u(x) - u(y)|^p}{|x - y|^{n+\sigma p}} dxdy \right)^{1/p}.$$

Für $s = m + \sigma$ definieren wir den *Sobolev-Slobodeckij-Raum* $W^{s,p}(\Omega)$ durch

$$W^{s,p}(\Omega) := \{u \in W^{m,p}(\Omega) : |D^\alpha u|_{\sigma,p} < \infty \quad \forall\, |\alpha| = m\},$$

wobei α ein Muliindex, $W^{m,p}(\Omega)$ ein gewöhnlicher Sobolev-Raum und $D^\alpha u$ die α-te schwache Ableitung von u ist. Weiterhin definieren wir eine Norm auf diesem Raum durch

$$||u||_{W^{s,p}(\Omega)} := (||u||^p_{W^{m,p}(\Omega)} + \sum_{|\alpha|=m} |D^\alpha u|^p_{\sigma,p})^{1/p}.$$

Mit dieser Norm ist $W^{s,p}$ ein separabler Banachraum, welcher im Falle $p > 1$ auch reflexiv ist.

Man sieht, dass in dieser Definition Räume für $s \in \mathbb{N}$ nicht definiert wurden. Aus diesem Grund verwenden wir im Falle $s \in \mathbb{N}$ die klassische Definition der Sobolevräume aus 1, womit die Sobolev-Slobodeckij-Räume für alle $s \in [0, \infty)$ definiert sind. Die Intuition hinter der definierten Halbnorm ist eine Quantifizierung der Regularität der Ableitungen der Ordnung m. In gewisser Hinsicht sind die Sobolev-Slobodeckij-Räume somit eine Analogie der Hölderräume $C^{m,\alpha}(\Omega)$ für schwach differenzierbare Funktionen, für Details siehe etwa [20]. Da die Einbettungssätze 5 wie erwähnt auch für beliebige Ordnungen $s \in [0, \infty)$ gültig sind, rechtfertigt sich auch die Bezeichnung *Interpolationsraum* für diese Art von Räumen. Zum Abschluss sei zu bemerken, dass die Sobolev-Slobodeckij-Räume nicht mit den Bessel'schen Potentialräumen zu verwechseln sind, welche auch oft die Bezeichnung $W^{s,p}(\Omega)$ tragen, und die klassichen Sobolevräume für gebrochene Ordnungen mittels Fourriermethoden verallgemeinern.

Wir kommen nun abschließend zu dem Spursatz, welcher die Auswertung von schwach differenzierbaren Funktionen auf einem Rand betrachtet. Dieser ist für uns deshalb von Interesse, da im Rahmen der Formoptimierung Auswertungen auf dem Rand, einer Form, natürlich auftreten. Er beschreibt, welches Maß an Regularität bei Auswertung auf dem Rand verloren geht. Die hier aufgeführte Variante betrachtet allgemeine $C^{m,1}$-Ränder, d.h. Ränder die sich als Graph einer Funktion mit Lipschitz-stetiger $m-ten$ Ableitung darstellen lassen, vgl. [20]. Wir beschränken uns auf den für unsere Zwecke ausreichenden Hilbertraumfall $p = 2$, für allgemeinere Varianten und Bedingungen hierzu, siehe etwa [1] oder [20].

Theorem 7 (Spursatz)**.** Sei $\Omega \subseteq \mathbb{R}^n$ offen und zusammenhängend mit $C^{m,1}$-Rand $\partial\Omega$. Zudem sei $s = m + \sigma > \frac{1}{2}$ mit $m \in \mathbb{N}$, $\sigma \in [0, 1)$. Dann gilt

(i) Es existiert ein eindeutiger, stetiger Spuroperator $S : H^s(\Omega) \to H^{s-\frac{1}{2}}(\partial\Omega)$, mit $S(u) = u_{|\partial\Omega}$ für alle $u \in C^\infty(\bar{\Omega})$.

(ii) Es existiert ein eindeutiger, stetiger Fortsetzungsoperator $F : H^{s-\frac{1}{2}}(\partial\Omega) \to H^s(\Omega)$ mit $S \circ F = \mathrm{id}_{H^{s-\frac{1}{2}}(\Omega)}$.

Der Spursatz (engl. *trace theorem*) besagt also unter anderem, dass bei Auswertung einer $H^s(\Omega)$-Funktion auf dem Rand eine halbe Ordnung verloren geht. Aus diesem Grunde werden in folgenden Kapiteln die Räume $H^{\frac{1}{2}}(\partial\Omega)$ als die natürlichen Räume für die meisten auf Formen definierten Funktionen auftauchen.
Zuletzt werden wir im Laufe der Arbeit noch hinreichend starke Regularität für Lösungen des Poisson-Problems benötigen. Aus diesem Grund führen wir hier noch den Satz über Regularität von schwachen Lösungen auf, vgl. [23] 2.13., wobei wir hier den Satz auf den Fall unseres Poisson-Problems spezialisieren.

Theorem 8 (höhere Regularität schwacher Lösungen)**.** Betrachte das Poisson-Problem 1.1 in schwacher Formulierung. Sei $m \in \mathbb{N}$, $f \in H^m(\Omega)$ und der Rand $\partial\Omega$ sei von der Ordnung C^{m+2}. Sei $y \in H_0^1(\Omega)$ die schwache Lösung des Poisson-Problems. Dann gilt $y \in H^{m+2}(\Omega)$.

Im allgemeineren Fall elliptischer Probleme gilt der Satz weiterhin, falls die Koeffizientenfunktionen $a^{i,j}, b^j, c \in C^{m+1}(\bar{\Omega})$ sind, vgl. die oben genannte Quelle, wobei $\bar{\Omega}$ wie üblich der Abschluss von Ω bezüglich der euklidischen Metrik ist. Fordert man keine Regularität des Randes $\partial\Omega$, so gilt die obige Aussage zwar nicht bis zum Rand, jedoch lokal im Sinne $y \in H^{m+2}_{loc}(\Omega)$.

Die von uns nun eingeführten Grundlagen der schwachen Lösungstheorie für die Poissongleichung reichen aus, um diese gewinnbringend in der Formoptimierung anzuwenden. Vor allem im dritten Kapitel, wo wir Formgradienten konstruieren, wird die Theorie starken Anklang finden. Zuvor werden wir im nächsten Kapitel eine Einführung in die Grundlagen der Formoptimierung geben.

2 Einführung in die theoretischen Grundlagen der Formoptimierung

In diesem Kapitel möchten wir eine Einführung in die grundlegenden Begriffe der Formoptimierung geben. Die Formoptimierung ist der Bereich des Optimal Control, welcher sich mit der Wahl einer, in gewisser Hinsicht, optimalen Form beschäftigt. Anders als in der klassischen PDE-constrained Optimization ist hier die Steuerung somit keine Funktion, sondern eine Form. Was man mathematisch präzise unter Formen versteht, werden wir im Laufe dieser Kapitels präzisieren. Zu dem Umgang mit Optimierungsproblemen von Formen gibt es in der Literatur verschiedene Ansätze.
Ein möglicher Ansatz zur Lösung ist die direkte Parametrisierung. Das zugrunde liegende Gebiet wird mit Hilfe von Kurven erzeugt, wobei diese von Parametern abhängig sind. Dadurch wird die Form mit Hilfe der endlich vielen Parameter gesteuert, nach denen im Anschluss optimiert werden kann. Der Vorteil dieser Methode ist, dass man sich in die Situation des Optimierens in endlichdimensionalen Vektorräumen zurückziehen kann, was erhebliche theoretische und numerische Erleichterungen mit sich bringt. Beispielhaft sei hier [11] genannt, wo die Autoren sogenannte B-Splines verwenden, um mit Hilfe eines adaptiv knotenerzeugenden Algorithmus in Kombination mit dem BFGS-Verfahren verbesserte Konvergenz zu erzielen. Ein großer Nachteil dieses Ansatzes ist, dass nicht beliebige Geometrien erreicht werden können, da diese wesentlich von der Wahl der parametrisierenden Kurven abhängen.
Ein weiterer Ansatz, den wir in dieser Arbeit weiter verfolgen werden, ist die Verwendung des sogenannten *Formkalküls* (engl. *shape calculus*), siehe beispielsweise [18] und [29]. Hierbei wird die Form nicht endlich parametrisiert, stattdessen findet die Optimierung im Raum aller Formen (engl. *shape space*) statt. Dies bietet den Vorteil, dass sich dieser Ansatz nicht künstlich restriktiv auf die Auswahl möglicher Geometrien auswirkt. Als Hindernisse bei diesem Ansatz treten eine Vielzahl von Phänomenen auf. Beispielsweise weist

der Raum aller Formen keine natürliche Vektorraumstruktur auf, weshalb die Optimierung in diesem Raum sich nicht auf Optimierung in Banach- oder Hilberträumen zurückführen lässt. Somit stellt sich die Frage nach einer sinnvollen Wahl der Metrik auf diesem Raum, sowie dem Umgang mit den so erzeugten Strukturen und deren numerische Ausnutzung. Wir werden im dritten Kapitel einige Einblicke in diese Strukturen geben, und diese in den anschließenden Kapiteln numerisch ausnutzen. Um nicht den Rahmen dieser Arbeit zu sprengen, verweisen wir für weiterführende Erläuterungen zu diesem Gebiet auf [15], [25], [23], und bedienen uns stattdessen lediglich der grundlegenden Begrifflichkeiten und einiger ausgewählter Konzepte. In diese möchten wir, weitestgehend bei den Grundlagen beginnend, einführen, und beginnen mit den Begriffen zur Definition von Formoptimierungsproblemen.
Bei der numerischen Optimierung werden Objekte hinsichtlich durch Zielfunktionale definierter Kriterien bewertet. Die Güte einer Form ist dabei, siehe [23] oder [27], gegeben durch den skalaren Wert bei Auswertung eines sogenannten Formfunktionals.

Definition 9 (Formfunktional)**.** Sei $\mathcal{D} \subset \mathbb{R}^n$ nicht leer und offen mit $d \in \mathbb{N}_+$. Weiterhin sei $\mathcal{O} \subseteq 2^{\mathcal{D}}$ die Menge aller offenen Teilmengen von $\mathcal{D}$. Dann heißt die Abbildung

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

Formfunktional und $\Omega \in \mathcal{O}$ *zulässiges Gebiet.*

In der Literatur wird die Menge $\mathcal{D}$ oft *Hold-all-Domain* genannt, siehe etwa die oben genannten Quellen. Alle Formen, die zur Optimierung in Betracht gezogen werden, lassen sich in die Hold-all-Domain einbetten.
Da die Menge aller zur Optimierung verwendeten Formen $\mathcal{O}$ aus Perspektive der Praxis zu groß ist, fordern wir einige Regularitätseigenschaften, wie etwa in [18] und [3], und passen diese unserer Situation an:

Definition 10 (reguläre Formen)**.** Sei $\Omega \in \mathcal{O}$ ein zulässiges Gebiet, $\varepsilon > 0$, $\mathrm{int}(\Omega)$ das Innere und $\partial\Omega$ der Rand von Ω. Falls gilt:

i) $\partial\Omega$ ist eine $(d-1)$-dimensionale Untermannigfaltigket des $\mathbb{R}^n$, und zu jedem Punkt $p \in \partial\Omega$ existieren offene Umgebungen $U \subseteq \mathbb{R}^{d-1}, V \subseteq \mathbb{R}^n$, mit einer zugehörigen Parametrisierung $\varphi : U \to \mathbb{R}^n$, und einer hinreichend differenzierbaren Erweiterung $h : V \to \mathbb{R}^n$, so dass lokal gilt: $h(x,0) = \varphi(x)$ und $h(x, x_d) \in \mathrm{int}(\Omega)$, falls $-\varepsilon < x_d < 0$.

ii) Ω ist zusammenhängend, beschränkt und besitzt einen Lipschitz-Rand.

Dann heißt Ω *reguläres Gebiet* und $\partial\Omega$ *reguläre Form.*

Anschaulich bedeutet die erste Forderung an Regularität, dass sich $\partial\Omega$ lokal so parametrisieren lässt, dass durch hinzufügen einer weiteren Koordinate x_d der umgebende Raum mit parametrisiert wird und anhand des Vorzeichens von x_d erkennbar ist, ob sich der zugehörige Punkt $h(x, x_d)$ im Inneren oder außerhalb von Ω befindet. Dies erleichtert die Beschreibung äußerer Normalen.
Nun möchten wir die Problemstellung der Formoptimierung im Rahmen dieser Arbeit definieren, in Analogie zu [24].

Definition 11 (Abstraktes Formoptimierungsproblem)**.** Sei $\mathcal{O}$ die Menge aller zulässigen Gebiete einer Hold-all-Domain $\mathcal{D} \subseteq \mathbb{R}^n$ und $\mathcal{J}$ ein auf dieser Menge beschränktes, wohldefiniertes Formfunktional. Weiterhin seien Y, Z Banachräume und $c : Y \times \mathcal{O} \to Z$ eine hinreichend glatte Abbildung. Dann heißt das Problem

$$\begin{aligned} &\min_{(y,\Omega)\in Y\times\mathcal{O}} \mathcal{J}(y,\Omega) \\ &\text{s.t. } c(y,\Omega) = 0 \end{aligned}$$

Problem der Formoptimierung.

Die Nebenbedingung $c(y, \Omega) = 0$ wird in dieser Arbeit in Form einer partiellen Differentialgleichung auf dem Gebiet Ω mit Lösung y auftreten, nämlich der in 1.1 eingeführten Poisson-Gleichung. Da $\mathcal{O}$ wie zuvor erwähnt keine Vektorraumstruktur besitzt, lässt sich $\mathcal{O}$ im Allgemeinen nicht als Hilbert- oder Banachraum auffassen, wodurch Formoptimierungsprobleme in dieser Formulierung aus struktureller Sicht im Allgemeinen deutlich komplexer sind als Optimierungsprobleme in Hilbert- oder Banachräumen. Für die Grundlagen dieser Theorie der Formoptimierung verweisen wir auf [23].
Der in Definition 10 geforderte Lipschitz-Rand ist sowohl aus theoretischer, als aus praktischer Sicht gerechtfertigt, da Existenz und Eindeutigkeit der im Anschluss betrachteten PDE auf Gebieten mit Lipschitz-Rand gesichert sind, und dieser somit für die Wohlgestelltheit des Optimierungsproblems notwendig ist.
Die Verfahren zur Optimierung von Formen bei partiellen Differentialgleichungen, die wir in dieser Arbeit betrachten, basieren auf der Differenzierbarkeit der obigen Ausdrücke, unter anderem bezüglich der Form Ω. Im Folgenden führen

wir die hierzu gehörigen Begriffe ein. Zunächst betrachten wir Deformationen von Gebieten in Analogie zu [27].

Definition 12 (Deformiertes Gebiet)**.** Sei $\Omega \in \mathcal{O}$ ein zulässiges Gebiet, $\tau > 0$, sei $\{T_t\}_{t\in[0,\tau]}$ eine Familie von bijektiven Abbildungen $T_t : \Omega \to \mathbb{R}^n$ mit $T_0 = id$. Dann heißt $\Omega_t := \{T_t(x)|x \in \Omega\}$ *deformiertes Gebiet*.

Zwei häufig Verwendung findende Arten solcher Familien von Deformationen $\{T_t\}_{t\in[0,\tau]}$ sind die *Pertubation of Identity* und die *Velocity method*, siehe [27]. Die *Pertubation of Identity* ist definiert über ein hinreichend differenzierbares Vektorfeld V durch

$$T_t : \Omega \to \mathbb{R}^n, x \mapsto x + tV(x).$$

Die Deformationen der *Velocity method* sind definiert durch den Fluss $T_t(x) := \xi(t, x)$, welcher gegeben ist durch das Anfangswertproblem

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

Für den Rest dieser Arbeit werden wir die Pertubation of Identity verwenden. Mit Hilfe der eingeführten Deformationen können wir nun die Formableitung nach [27] definieren.

Definition 13 (Formableitung)**.** Sei $\Omega \in \mathcal{O}$ ein reguläres Gebiet, $V \in C^1(\Omega, \mathbb{R}^n)$ ein differenzierbares Vektorfeld, und $\mathcal{J} : \mathcal{O} \to \mathbb{R}$ ein Formfunktional. Falls der Grenzwert

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

existiert, und die Abbildung

$$D\mathcal{J}(\Omega)[\cdot] : C^1(\Omega, \mathbb{R}^n) \to \mathbb{R}, V \mapsto D\mathcal{J}(\Omega)[V]$$

stetig und linear für alle $V \in C^1(\Omega, \mathbb{R}^n)$ ist, so heißt $D\mathcal{J}(\Omega)[V]$ *Formableitung* von $\mathcal{J}$ in Ω in Richtung V.

Es ist denkbar, die Richtungen V etwas allgemeiner zu wählen, indem man H^1-Vektorfelder als Ableitungsrichtung zulässt. Dies hängt mit der nötigen Anwendung des Transformationssatzes bei der Formableitung zusammen, dessen Anwendung durch Voraussetzungen an die Form und Ableitungsrichtung

gesichert sein sollte. Die obige Allgemeinheit wird für unsere Ziele völlig ausreichen.
Die Formableitung besitzt zwei äquivalente Formulierungen, die sogenannte *Volumenformulierung*, gekennzeichnet durch Ω, und die *Randformulierung*, gekennzeichnet durch $\Gamma := \partial\Omega$, siehe [27]:

$$\begin{aligned} D\mathcal{J}_\Omega(\Omega)[V] &= \int_\Omega F(x)V(x)dx \\ D\mathcal{J}_\Gamma(\Omega)[V] &= \int_\Gamma f(s)\langle V(s), n(s)\rangle ds, \end{aligned} \tag{2.1}$$

wobei $F(x)$ ein (Differential-) Operator ist, welcher linear auf das Richtungsvektorfeld V wirkt, sowie einer Funktion $f : \Gamma \to \mathbb{R}$. Die Existenz einer zur Volumenform äquivalenten Randformulierung ist gesichert durch den *Hadamard'schen Darstellungssatz*. Dieser liefert im Allgemeinen die Existenz einer Distribution f, so dass Volumen- und Randformulierung der Formableitung äquivalent sind, für die genaue Aussage, siehe [23], Theorem 4.7. Wir werden in dieser Arbeit stets voraussetzen, dass genügend Regularität vorhanden ist, so dass $D\mathcal{J}$ in der Tat mittels einer Funktion f auf dem Rand darstellbar ist. Die hier definierte Formableitung ist im Englischen auch bekannt als *Eulerian Semiderivative* oder *Lie Semiderivative*. Es gibt weitere äquivalente Varianten die Formableitung zu definieren, hierzu siehe [18]. Oft von zentraler Schwierigkeit in der theoretischen Behandlung von Formoptimierungsproblemen ist der Nachweis der Formdifferenzierbarkeit eines Formfunktionals. Hierzu gibt es eine Vielzahl von verschiedenen Techniken, beispielsweise der Min-Max Formulierung von Correa und Seeger, Céa's klassischer Lagrange-Methode oder der Methoden mittels Materialableitung. Da wir diese nicht explizit verwenden werden, sondern die Resultate aus entsprechenden Quellen zitieren werden, führen wir nicht in diese ein, und verweisen den interessierten Leser auf [18], wo diese in einheitlicher Notation geordnet zusammengetragen sind.
Wir kommen nun zu einer, zwar immer noch abstrakt gehaltenen, jedoch konkreter anwendungsbezogenen Formulierung von Formoptimierungsproblemen, vgl. [24], Abschnitt 2.

Definition 14 (PDE beschränktes Formoptimierungsproblem)**.** Sei $\mathcal{J}$ ein Formfunktional, sei $\Omega \in \mathcal{O}$ eine zulässige Form aus einer Hold-all-Domain $\mathcal{D} \subset \mathbb{R}^n$. Weiterhin sei $y \in H(\Omega)$ eine Funktion aus dem von Ω abhängenden Hilbertraum $H(\Omega)$. Betrachte eine von Ω abhängige Bilinearform $a_\Omega : H(\Omega) \times H(\Omega) \to \mathbb{R}$ und eine ebenso von Ω abhängige Linearform $b_\Omega : H(\Omega) \to \mathbb{R}$.

Dann heißt das beschränkte Minimierungsproblem

$$\begin{aligned} &\min_{y,\Omega} \mathcal{J}(y,\Omega) \\ &\text{s.t. } a_\Omega(y,p) = b_\Omega(p) \quad \forall p \in H(\Omega) \end{aligned} \tag{2.2}$$

PDE beschränktes Formoptimierungsproblem (engl.*PDE-constrained shape-optimization problem*).

Mit der Gleichungsnebenbedingung deuten wir direkt an die Variationsformulierung partieller Differentialgleichungen an. Außerdem haben wir diese Definition aus [24], Definition 2.1, etwas abgewandelt, um eine Einführung in Begriffe der Bündeltheorie zu vermeiden. Das wird uns bequemer ermöglichen, weitere Definitionen und Techniken entsprechend anzugeben, womit wir direkt beginnen.

Definition 15 (Lagrangefunktion)**.** Betrachte ein PDE beschränktes Formoptimierungsproblem. Sei $\Omega \in \mathcal{O}$ eine reguläres Gebiet, $H(\Omega)$ ein von diesem Gebiet abhängiger Hilbertraum. Weiterhin seien $y, p \in H(\Omega)$. Dann heißt die Funktion

$$\mathcal{L}(y,\Omega,p) := \mathcal{J}(y,\Omega) + a_\Omega(y,p) - b_\Omega(p)$$

Lagrangefunktion für das gegebene PDE beschränkte Formoptimierungsproblem.

Diese Lagrangefunktion ermöglicht es uns, Optimalitätskriterien für das PDE beschränkte Formoptimierungsproblem aufzustellen. Ein problematischer, und gleichzeitig vorteilhafter Aspekt des rein formalen Lagrangeansatzes ist, dass die Regularität der Funktionen, für welche die Lagrangefunktion definiert ist, im Vorfeld nicht bekannt sein muss oder ist. Aus diesem Grunde ist eine ex-post Analyse der zur konsistenten Beschreibung nötigen Funktionenräume wichtig. Wir fahren fort mit notwendigen Optimalitätskriterien für Lösungen, welche gegeben werden in Form von Variationsformulierungen, siehe [24], Definition 2.1, wobei dort hinreichend glatte Richtungen V verwendet werden.

Definition 16 (Adjungierte und Designgleichung)**.** Betrachte ein PDE beschränktes Formoptimierungsproblem mit differenzierbarem Formfunktional $\mathcal{J}$. Verwende die Notation von 15 und bezeichne mit $D\mathcal{J}$ die zu $\mathcal{J}$ gehörige Formableitung. Seien $\tilde{y} \in H(\tilde{\Omega})$ und $\tilde{\Omega} \in \mathcal{O}$ optimale Lösungen des Problems 2.2. Dann gilt die Variationsgleichung

$$a_{\tilde{\Omega}}(\tilde{y}, z) = -\frac{\partial}{\partial y}\mathcal{J}(\tilde{y}, \tilde{\Omega})(z) \quad \forall z \in H(\tilde{\Omega}), \tag{2.3}$$

welche wir als *adjungierte Gleichung* bezeichnen. Sei $\tilde{p} \in H(\tilde{\Omega})$ die Lösung der adjungierten Gleichung 2.3. Dann gilt weiterhin die Variationsgleichung

$$D\mathcal{L}(\tilde{y}, \tilde{\Omega}, \tilde{p})[V] = 0 \quad \forall V \in C^1(\tilde{\Omega}, \mathbb{R}^n), \tag{2.4}$$

welche wir als *Design Gleichung* bezeichnen werden, wobei $\mathcal{L}$ die zugehörige Lagrangefunktion wie in 15, und $D\mathcal{L}$ die zugehörige Formableitung ist. Außerdem gilt

$$a_{\tilde{\Omega}}(\tilde{y}, p) = b_{\tilde{\Omega}}(p) \quad \forall p \in H(\tilde{\Omega}), \tag{2.5}$$

wobei diese Gleichung, welche als Nebenbedingung in 2.2 auftritt, *Zustandsgleichung* (*engl. state equation*) genannt wird.

Wie man unschwer erkennt, entstehen die notwendigen Bedingungen aus den Ableitungen der Lagrangefunktion 15 nach dem Zustand y, dem Gebiet Ω und dem adjungierten Zustand p. Fasst man diese Bedingung in einer Gleichung zusammen, so erhält man das KKT-System

$$D\mathcal{L}(\tilde{y}, \tilde{\Omega}, \tilde{p}) \begin{pmatrix} h_y \\ h_\Omega \\ h_p \end{pmatrix} = 0 \quad \forall h_\Omega \in C^1(\tilde{\Omega}, \mathbb{R}^n)\ \forall h_y, h_p \in H(\tilde{\Omega}) \tag{2.6}$$

als notwendige Bedingung zur Lösung von 2.2, siehe [24], wobei wir die dortige Formulierung mit Hilfe von Vektorbündeln auf die hier angepasste Notation übersetzt haben, was aufgrund der Definition des Tangentialbündels mittels von Formen abhängiger Kreuzprodukte von Hilberträumen möglich ist. An dieser Stelle wäre es möglich einen Lagrange-Newton Ansatz durchzuführen, und ein Verfahren zur Lösung des Problems 2.2 mit quadratischer Konvergenz zu gewinnen, was die Autoren von [24] getan haben. Diesen führen wir kurz im nächsten Kapitel unter 3.15 auf.
Wir führen jetzt noch eine technische Notation zur Beschreibung von Sprüngen auf Rändern ein, vgl. [24], Abschnitt 3, woraufhin wir unser Modellproblem definieren.

Definition 17. Sei $\Omega \subset \mathcal{D}$ ein reguläres Gebiet einer Hold-all-Domain $\mathcal{D}$. Dann definieren wir das *Sprungsymbol* $[[\cdot]]$, für eine auf $\mathcal{D}$ definierte Funktion f, auf dem Rand $\partial\Omega$ durch

$$[[f]] = f_{|\mathcal{D}\setminus\Omega} - f_{|\Omega}.$$

Die obigen Definition ist dabei so zu verstehen, dass jeweils Grenzwerte von innerhalb, und von außerhalb Ω gebildet werden, und diese dann voneinander

abgezogen werden, sofern diese existieren. Die Existenz eines sinnvollen Inneren und Äußeren ist durch die Regularität des Gebietes gewährleistet.
Wir besitzen nun den Apparat zur Formulierung unseres Modellproblems und zugehöriger Optimalitätsbedingungen und Ableitungen. Hierzu wählen wir wie vorweggenommen das Poisson-Problem 1.1, für welches wir ausreichende Theorie in Kapitel 1 eingeführt haben. Das Problem findet sich in ähnlicher Art in [23], 4.2., sowie in [24], unter Remark 2, wobei wir die Notation nicht von der Wahl des Gebietes Ω abhängen lassen, dies aber implizit meinen.

Definition 18 (Modellproblem)**.** Sei $\mathcal{D} := (0,1)^2 \subset \mathbb{R}^2$ das Einheitsquadrat im $\mathbb{R}^2$, welches als Hold-all-Domain fungiert. Weiterhin sei $\Omega \subset (0,1)^2$ ein reguläres Gebiet in der Hold-all-Domain mit zugehöriger Form $\partial\Omega$, und eine stückweise konstante Funktion $f \in \mathcal{L}^2((0,1)^2)$ der Art

$$\begin{aligned} f(x) = f_1 \in \mathbb{R} \quad \forall x \in (0,1)^2 \setminus \Omega \\ f(x) = f_2 \in \mathbb{R} \quad \forall x \in \Omega. \end{aligned}$$

Sei $\hat{y} \in \mathcal{L}^2((0,1)^2)$ und $\nu > 0$. Dann heißt das nun folgende Problem *Modellproblem*:

$$\begin{aligned} \min_{\Omega \in \mathcal{O}} \mathcal{J}(\Omega) : = \frac{1}{2}\int_{\mathcal{D}} (y-\hat{y})^2 dx + \nu \int_{\partial\Omega} 1 ds \\ \text{s.t.} -\Delta y = f \quad \text{in } \mathcal{D} \\ y = 0 \quad \text{auf } \partial\mathcal{D}. \end{aligned} \tag{2.7}$$

Zudem definieren wir für 2.7 explizit eine sogenannte *Interface Condition*

$$[[y]] = 0, \quad \left[\left[\frac{\partial y}{\partial n}\right]\right] = 0 \quad \text{auf } \partial\Omega, \tag{2.8}$$

wobei n das äußere Einheitsnormalenvektorfeld auf $\partial\Omega$ ist. Zusammen erhalten wir die schwache Formulierung der Zustandsgleichung

$$\int_{\mathcal{D}} \nabla y^T \nabla p \, dx - \int_{\partial\Omega} \left[\left[\frac{\partial y}{\partial n} p\right]\right] ds = \int_{\mathcal{D}} fp \, dx \quad \forall p \in H_0^1(\mathcal{D}). \tag{2.9}$$

Weiterhin besitzt das Modellproblem die, nach 2.3 definierte, adjungierte Gleichung

$$\begin{aligned} -\Delta p = -(y-\hat{y}) \quad \text{in } \mathcal{D} \\ p = 0 \quad \text{auf } \partial\mathcal{D} \\ [[p]] = 0 \quad \text{auf } \partial\Omega \\ \left[\left[\frac{\partial p}{\partial n}\right]\right] = 0 \quad \text{auf } \partial\Omega. \end{aligned} \tag{2.10}$$

Wie wir sehen, besteht das Zielfunktional $\mathcal{J}$ aus zwei Komponenten. Der erste Summand bildet das eigentliche Funktional von Interesse, welches wir von nun an mit $\mathcal{J}_{target}$ bezeichnen werden, wobei die optimale Form $\partial\tilde{\Omega}$ möglichst einen Zustand $\tilde{y}$ erzeugen soll, welcher einem gegebenen Sollzustand $\hat{y}$ entspricht. Der zweite Summand

$$\nu \int_{\partial\Omega} 1 ds =: \mathcal{J}_{reg}(\Omega)$$

wird gemeinhin als *Perimeter-Regularisierung* bezeichnet, wobei $\nu > 0$. Diese hat den Nutzen, Regularität und somit Existenz und Eindeutigkeit des Modellproblems 2.7 zu gewährleisten. Ohne die Regularisierung wären beispielsweise entartete Formen, deren Rand als Funktion unendlicher Variation darstellbar sind, zugelassen, siehe zum Beispiel die angegebene Quelle bei [24], unter Remark 2.
Die Zustandsgleichung ist ein gewöhnliches Poisson-Problem mit Dirichlet-Randwerten, welches wir hier symbolisch in starker Form notiert haben. Die starke Notation repräsentiert hier also die Variationsformulierung des Poisson-Problems mit Dirichlet-Randwerten, wobei gleiches für die adjungierte Gleichung gilt. Für die Herleitung der adjungierten Gleichung, welche mittels 2.3 erfolgt, verweisen wir auf [23], 4.2.1. Die Existenz einer Lösung der Zustands- als auch adjungierten Gleichung sind gesichert durch das in 4 eingeführte Lemma von Lax-Milgram.
Die Interface Condition 2.8 sorgt bei uns dafür, dass der Zustand y stetig vom inneren Gebiet Ω zum äußeren Gebiet $\mathcal{O} \setminus \Omega$ verläuft, sowie, dass der Fluss $\frac{\partial y}{\partial n}$ auch stetig ist. Ohne die Interface Condition wäre dies für allgemeine Probleme nicht der Fall, in unserem Falle wäre jedoch das Weglassen dieser wegen Regularität der Lösung möglich.
Damit wir Wohldefiniertheit von im folgenden Kapitel 3.1 definierten Metriken für Formräume sicherstellen können, benötigen wir, dass wir eine Einschränkung der Lösung der Zustandsgleichung auf den Rand mit $y_{|\partial\Omega} \in H^{1/2}(\partial\Omega)$ besitzen. Mit unseren Voraussetzungen $f, \bar{y} \in \mathcal{L}^2((0,1)^2)$, sowie einem hinreichend glatten Rand $\partial\Omega$, erhalten wir $y \in H^2((0,1)^2)$ nach dem Satz 8 für höhere Regularität der Lösung. Kombiniert man dies mit den auf Sobolev-Slobodeckij-Räumen verallgemeinerten Spursatz 7, so erhalten wir wie gewünscht $y_{|\partial\Omega} \in H^{1/2}(\partial\Omega)$. Ob die hinreichend glatten Ränder in der Theorie und Praxis gerechtfertigt sind, ist wie schon erwähnt eine gänzlich andere wichtige Diskussion, welche wir in dieser Arbeit nicht führen möchten.
Wir können nun für unser Modellproblem 2.7 die Formableitung angeben. Die genaue Herleitung, die das Theorem von Correa-Seeger verwendet und auf Ma-

terialableitungen aufbaut, findet sich in [23], die alternative Randformulierung für die Formableitung der Perimeter-Regularisierung findet sich in [5].

Theorem 19 (Formableitung für das Modellproblem)**.** Betrachte das Modellproblem 2.7 mit Interface Condition. Sei $y \in H_0^1((0,1)^2)$ Lösung der Zustandsgleichung aus 2.7 und $p \in H_0^1((0,1)^2)$ Lösung der adjungierten Gleichung 2.10. Dann hat das Zielfunktional aus 2.7 *ohne* Perimeter-Regularisierung $\mathcal{J}_{target}$ folgende Volumenformulierung für alle $\Omega \in \mathcal{O}$ und Richtungen $V \in C^1(\mathcal{D}, \mathbb{R}^2)$:

$$\begin{aligned} D\mathcal{J}_{target}(\Omega)[V] = \int_{\mathcal{D}} & -\nabla y^T(\nabla V + \nabla V^T)\nabla p - pV^T\nabla f \\ & +\mathrm{div}(V)\left(\frac{1}{2}(y-\bar{y})^2 + \nabla y^T\nabla p - fp\right) dx. \end{aligned} \tag{2.11}$$

Falls sogar genügend Regularität vorhanden ist, so dass $y \in H^2((0,1)^2)$ und $p \in H^2((0,1)^2)$, so besitzt die Formableitung die Randformulierung

$$D\mathcal{J}_{target}(\Omega)[V] = -\int_{\partial\Omega} [[f]]p\langle V, n\rangle ds. \tag{2.12}$$

Zusammen mit der Perimeter-Regularisierung erhält man

$$D\mathcal{J}_{target}(\Omega)[V] = \int_{\partial\Omega} (-[[f]]p + \nu\kappa)\langle V, n\rangle ds \tag{2.13}$$

wobei $\kappa := \mathrm{div}_{\partial\Omega}(n)$ die mittlere Krümmung von $\partial\Omega$ definiert über die tangentiale Divergenz ist. Außerdem besitzt die Perimeter-Regularisierung alternative Randformulierung

$$D\mathcal{J}_{reg}(\Omega)[V] = \nu\int_{\partial\Omega} \mathrm{div}(V) - \langle\frac{\partial V}{\partial n}, n\rangle ds, \tag{2.14}$$

wobei n erneut das äußere Einheitsnormalenvektorfeld von $\partial\Omega$ ist.

Diese Formableitungen lassen sich nun in ableitungsbasierten Optimierungsmethoden verwenden. Wie man sieht, muss man zum Auswerten der Formableitung in einer bestimmten Form $\partial\Omega$ die zugehörige Zustandsgleichung bei 2.7, und die entsprechende adjungierte Gleichung 2.3 lösen. Weiter ließe sich dann aus der Ableitungsinformation ein Gradient erzeugen, mit dessen Hilfe man beispielsweise ein Verfahren des steilsten Abstiegs einsetzen kann. Um einen

solchen Gradienten zu definieren, werden wir uns im folgenden Kapitel Gedanken machen, welche zugrundeliegende Metrik wir auf dem Raum aller Formen annehmen, um mit dieser einen Gradienten in Anlehnung an den Riesz'schen Darstellungssatz zu definieren. An dieser Stelle sei erwähnt, dass wir hier Gebrauch von der geschlossenen Darstellung der Formableitung machen, welche wir in späteren Kapiteln implementieren werden. Die Formulierung der Formableitung der Perimeter-Regularisierung, welche ohne Krümmungsterm κ auskommt, wird von uns bevorzugt implementiert, da dies die Berechnung der Krümmung vermeidet, und somit eine Verallgemeinerung unseres Programms auf 3 Dimensionen erleichtert.
Ist die Formableitung bei einem gegebenen Problem nicht geschlossen angegeben, so lässt sich diese häufig numerisch mittels Techniken des *automatischen Differenzierens* gewinnen. Diese Methoden, die aus der Numerik für Differentialgleichungen bekannt sind, lassen sich auch auf Formableitungen verallgemeinern. Dies tut beispielsweise der Autor von [13], um auf diesem Wege Hesse-Matrizen zu erzeugen. Hierzu wird die sogenannte *Unified Form Language (UFL)* in der Programmiersprache Python verwendet, welche auch wesentlicher Bestandteil des auch von uns benutzt werdenden Programmpakets FEniCS ist, für Details, siehe etwa die Quellen bei [7], S.25.

3 BFGS-Algorithmus in der Formoptimierung

3.1 Shape Spaces und Formgradienten

Nachdem wir in die Grundlagen der Formoptimierung, möchten wir in diesem Abschnitt weiter auf den Grundlagen aufbauen und Rahmenbedingungen schaffen, unter welchen das endlichdimensionale BFGS-Verfahren in den Kontext von Formen abstrahiert werden können. Dies ist nicht ohne weiteres möglich, da a priori nicht klar ist, wie Gradienten von Formen definiert werden sollen, insbesondere deshalb, weil wir uns in unendlichdimensionalen *Formräumen (engl. shape-spaces)* befinden. Erschwert wird dies weiter dadurch, dass, wie eingangs bemerkt, keine natürliche Vektorraumstruktur auf dem betrachteten Formräumen vorhanden ist. Ziel dieses Abschnittes wird es sein zunächst die Wahl einer geeigneten Metrik im Sinne der Riemannschen Geometrie zu treffen, wodurch die Definition eines Gradienten überhaupt erst möglich wird. Im $\mathbb{R}^n$ würden wir zur Darstellung solcher Gradienten stets stillschweigend die euklidische Metrik verwenden, diese lässt sich jedoch offensichtlich nicht einfach im Shape-setting verwenden. Anschließend übertragen wir den BFGS-

Algorithmus auf die Formoptimierung. In diesem Abschnitt halten wir uns vor allem an [27], sowie an [23] und [17].
Zunächst definieren wir, was wir unter dem Raum aller Formen verstehen, vgl. [17]. Hierzu bleiben wir in zwei Dimensionen, da hier schon wesentliche Elemente und Zusammenhänge klar werden. Prinzipiell ist ein betrachten von höherdimensionalen Objekten auch möglich, sofern unter anderem die zugrundeliegende Topologie der Formen beachtet wird.

Definition 20 (Formraum für den $\mathbb{R}^2$)**.** Bezeichne mit $\mathrm{Emb}(S^1, \mathbb{R}^2)$ die Menge aller C^∞-Einbettungen von S^1 in den $\mathbb{R}^2$, und mit $\mathrm{Diff}(S^1)$ die Menge aller Diffeomorphismen von S^1 in sich selber. Dann heißt der Quotientenraum

$$B_e(S^1, \mathbb{R}^2) := \mathrm{Emb}(S^1, \mathbb{R}^2)/\mathrm{Diff}(S^1)$$

Formraum für den $\mathbb{R}^2$.

Man sieht, dass die Elemente von $B_e(S^1, \mathbb{R}^2)$ Äquivalenzklassen sind. In ihnen sind jeweils unter anderem Umparametrisierungen der selben geschlossenen C^∞-Kurven $c : \mathbb{R} \to \mathbb{R}^2$ enthalten. Das bedeutet, dass ein Punkt in $B_e(S^1, \mathbb{R}^2)$ als eine geschlossene, geometrische Kurve im $\mathbb{R}^2$ interpretiert werden kann. Betrachtet man nun die durch eine beschränkte Menge $\mathcal{D} \subset \mathbb{R}^2$ mit Lipschitz-Rand definierte Hold-all-Domain, so lassen sich Ränder $\partial\Omega$ von kompakten, beschränkten, zusammenhängenden Mengen $\Omega \subset \mathcal{D}$ mit C^∞-Rand genau mit solchen geschlossenen Kurven identifizieren. Zudem gilt außerdem, dass $B_e(S^1, \mathbb{R}^2)$ eine Mannigfaltigkeit bildet, siehe [27], was wesentlich von der Glattheit der Einbettungen abhängt.
Wir fahren fort mit unserer Konstruktion, und geben hier eine Darstellung des Tangentialbündels auf $B_e(S^1, \mathbb{R}^2)$ an. Wir verwenden hier explizit die Strukturen des $\mathbb{R}^2$ und $B_e(S^1, \mathbb{R}^2)$ für die Darstellung der Tangentialräume, wobei es sich bei unserer Definition um zu den üblichen Tangentialräumen der Differentialgeometrie isomorphe Objekte handelt. Für eine Definition der klassischen Tangentialräume, sowie eine tiefgreifende Einführung in die Differentialgeometrie, empfehlen wir [6], Kapitel 3.

Definition 21 (Tangentialbündel)**.** Sei $B_e(S^1, \mathbb{R}^2)$ der Formraum für den $\mathbb{R}^2$. Betrachte einen Repräsentant $c : S^1 \to \mathbb{R}^2$ eines Punktes in $B_e(S^1, \mathbb{R}^2)$, sowie das zugehörige äußere Einheitsnormalenvektorfeld n der mit der geschlossenen Kurve c identifizierten Form $\partial\Omega$. Dann gilt für den *Tangentialraum* $T_c B_e$

$$T_c B_e \cong \{h : h = \alpha n \text{ für } \alpha \in C^\infty(\Omega, \mathbb{R})\}.$$

Diese Darstellung gilt, da die Quotientenstruktur von B_e die Isomorphie der Immersionen von S^1 nach $\mathbb{R}^2$ mit den C^∞-Funktionen von S^1 nach $\mathbb{R}^2$ vererbt, für Details siehe [23], Kapitel 3. Somit ermöglicht diese isomorphe Darstellung es uns, Tangentialvektoren $v \in T_c B_e$ mit Hilfe von $C^\infty(\Omega, \mathbb{R})$-Funktionen zu beschreiben. Das nutzen wir aus, um auf B_e eine für unsere Zwecke geeignete Riemannsche Metrik zu definieren.
Hierzu gibt es verschiedene Möglichkeiten, je nachdem auf welche künftige Anwendung man abzielt. Nach [23] scheint eine Sobolev-Metrik, definiert mittels des Laplace-Beltrami-Operators, für unser Modellproblem natürlich zu erscheinen. Der Nachteil ist, das erheblicher Rechenaufwand beim Ermitteln des Gradienten aus einer Formableitung entsteht. Aus diesem Grund führen die Autoren von [27] eine Riemannsche Metrik auf Grundlage der sogenannten Dirichlet-zu-Neumann Abbildung ein, weshalb wir uns Ausführungen zu Sobolev-Metriken an dieser Stelle sparen, und für diese auf [17] und [23] verweisen. Im Folgenden führen wir eine sogenannte *Steklov-Poincaré-Metrik* ein, und verweisen für die hierzu nötigen Grundlagen der Differentialgleichungen und Sobolev-Slobodeckij-Räumen auf das Kapitel 1. Zunächst definieren wir die nötigen Abbildungen, vgl. [27].

Definition 22 (Verallgemeinerte Spurabbildung)**.** Sei $\mathcal{D} \subset \mathbb{R}^n$ eine beschränktes, offenes Gebiet. Sei $\Omega \subset \mathcal{D}$ zusammenhängend, offen und habe einen Lipschitz-Rand, und $\partial\Omega$ sei eine Form in diesem Gebiet. Dann heißt die Abbildung

$$\begin{aligned} \gamma : H_0^1(\mathcal{D}, \mathbb{R}^n) &\to H^{1/2}(\mathcal{D}, \mathbb{R}^n) \times H^{-1/2}(\mathcal{D}, \mathbb{R}^n) \\ U &\mapsto \begin{pmatrix} U_{|\partial\Omega} \\ \frac{\partial}{\partial n} U_{|\partial\Omega} \end{pmatrix} =: \begin{pmatrix} \gamma_0 U \\ \gamma_1 U \end{pmatrix}, \end{aligned}$$

verallgemeinerte Spurabbildung, wobei $\frac{\partial}{\partial n} U_{|\partial\Omega}$ die Ableitung von U auf der Form $\partial\Omega$ in Richtung des äußeren Einheitsnormalenvektorfeldes n ist.

Wir erinnern an dieser Stelle, dass diese Abbildung aufgrund der verallgemeinerten Spursätze 7 wohldefiniert ist. Nun geben wir uns eine symmetrische, koerzive Bilinearform $a : H^1(\mathcal{D}, \mathbb{R}^n) \times H^1(\mathcal{D}, \mathbb{R}^n) \to \mathbb{R}$ vor. Diese werden wir später gemeinsam mit der zu definierenden Riemannschen Metrik verwenden, um aus gegebenen Formableitungen Gradienten zu konstruieren. Gegeben einer solchen Bilinearform a definieren wir die zugehörigen Lösungsoperatoren zu folgendem Variationsproblem.

Definition 23 (Lösungsoperatoren)**.** Sei $\mathcal{D} \subset \mathbb{R}^n$ offen, zusammenhängend, beschränkt, und sei $\Omega \subset \mathcal{D}$ offen, zusammenhängend und habe einen C^∞-Rand $\partial\Omega$. Weiterhin sei $a : H^1(\mathcal{D}, \mathbb{R}^n) \times H^1(\mathcal{D}, \mathbb{R}^n) \to \mathbb{R}$ eine koerzive, symmetrische Bilinearform. Dann heißt der Operator

$$\begin{aligned} E_N : H^{-1/2}(\partial\Omega, \mathbb{R}^n) &\to H_0^1(\mathcal{D}, \mathbb{R}^n) \\ u &\mapsto U, \end{aligned}$$

Neumann-Lösungsoperator, wobei U die Lösung des Problems

$$a(U, V) = \int_{\partial\Omega} u^T(\gamma_0 V) ds \quad \forall V \in H_0^1(\mathcal{D}, \mathbb{R}^n)$$

mit der dualen Paarung $u^T(\gamma_0 V)$ ist. Außerdem heißt der Operator

$$\begin{aligned} E_D : H^{1/2}(\partial\Omega, \mathbb{R}^n) &\to H_0^1(\mathcal{D}, \mathbb{R}^n) \\ u &\mapsto U, \end{aligned}$$

Dirichlet-Lösungsoperator, wobei U die Lösung des Problems

$$\begin{aligned} a(U, V) &= 0 \\ \text{unter } U_{|\partial\Omega} &= u \end{aligned} \quad \forall V \in H_0^1(\mathcal{D}, \mathbb{R}^n),$$

ist.

Wir fahren fort und definieren die projizierten Neumann-zu-Dirichlet und Dirichlet-zu-Neumann-Operatoren, welche wir zur Konstruktion der Riemannschen Metrik benötigen, siehe [27].

Definition 24 (projizierte Randwertoperatoren)**.** Seien die Voraussetzungen von Definition 22 und 23 gegeben. Dann heißt der Operator

$$\begin{aligned} S^p : H^{-1/2}(\partial\Omega) \to H^{-1/2}(\partial\Omega, \mathbb{R}^n) \to H_0^1(\mathcal{D}, \mathbb{R}^n) \to H^{1/2}(\partial\Omega, \mathbb{R}^n) \to H^{1/2}(\partial\Omega) \\ \alpha \mapsto n^T[\gamma_0 \circ E_N(\alpha \cdot n)] \end{aligned}$$

projizierter Neumann-zu-Dirichlet-Operator, wobei n^T eine duale Paarung meint. Der Operator

$$\begin{aligned} T^p : H^{1/2}(\partial\Omega) \to H^{1/2}(\partial\Omega, \mathbb{R}^n) \to H_0^1(\mathcal{D}, \mathbb{R}^n) \to H^{-1/2}(\partial\Omega, \mathbb{R}^n) \to H^{-1/2}(\partial\Omega) \\ \alpha \mapsto n^T[\gamma_1 \circ E_D(\alpha \cdot n)] \end{aligned}$$

heißt *projizierter Dirichlet-zu-Neumann-Operator*.

Die hier definierten Operatoren sind nicht die klassischen Dirichlet-zu-Neumann-Operatoren, welche auch *Steklov-Poincaré-Operatoren* genannt werden, da diese nicht in projizierter Variante definiert werden. Was die Operatoren im Wesentlichen tun, ist für gegebene Dirichlet- bzw. Neumann-Bedingung die Lösung des durch die Bilinearform a definierten Problems zu finden, und die entsprechende andere, automatisch konsistente, Neumann- bzw. Dirichlet-Bedingung zurückzugeben. Das heißt, für gegebene Dirichlet-Bedingung liefert der projizierte Dirichlet-zu-Neumann-Operator die Neumann-Bedingung, welche die selbe Lösung erzeugt, und vice versa.
Die klassischen Steklov-Poincaré-Operatoren sind zueinander invers, beide koerziv, symmetrisch in dualer Paarung und stetig, siehe [27], unter Definition 3.1. Die projizierten Operatoren besitzen weiterhin alle genannten Eigenschaften, außer der des zueinander Inversen, denn es gilt im Allgemeinen $T^p \neq (S^p)^{-1}$. Um Rechenaufwand einzusparen, den Implementierungsaufwand zu verringern, und Glättungseigenschaften auszunutzen, fällt die Wahl des Operators zur Definition des Skalarproduktes auf den Tangentialräumen auf $(S^p)^{-1}$, für Details, unter anderem zu spektraler Äquivalenz der Kandidaten, siehe [27] und die dort genannten Quellen. Wir kommen nun zu unserem Etappenziel, eine Riemannsche Metrik auf dem Formraum für den $\mathbb{R}^2$ zu definieren, vgl. [27].

Definition 25 (Steklov-Poincaré-Metrik)**.** Seien die Voraussetzungen wie in 24. Dann heißt das Skalarprodukt

$$g^S : H^{1/2}(\partial\Omega) \times H^{1/2}(\partial\Omega) \to \mathbb{R}$$
$$(\alpha, \beta) \mapsto \langle \alpha, (S^p)^{-1}\beta \rangle = \int_{\partial\Omega} \alpha(s) \cdot ((S^p)^{-1}\beta)(s) ds,$$

Steklov-Poincaré-Metrik, wobei die Produkte auf der rechten Seite als duale Paarungen zu verstehen sind.

Das so definierte Skalarprodukt ist auf allgemeinen Sobolev-Slobodeckij-Räumen definiert, da Lösungen der Zustandsgleichung bei Auswertung auf dem Rand in der Regel Ordnung 1/2 besitzen, siehe Kapitel 1. Diese Räume enthalten, wie wir in der Einführung zu Differentialgleichungen gezeigt haben, die C^∞-Funktionen auf dem Rand. Somit ist das Skalarprodukt für Tangentialvektoren aller Formen aus $B_e(S^1, \mathbb{R}^2)$ wohldefiniert, da wir erneut Tangentialvektoren wie in der Definition 21 mit den Koeffizientenfeldern der äußeren Normalenvektorfelder identifizieren.
Wir kommen nun zu dem Zusammenhang des in Kapitel 2 definierten Formkalküls und der soeben konstruierten Riemannschen Mannigfaltigkeit $(B_e(S^1, \mathbb{R}^2), g^S)$.

Für die in dieser Arbeit genutzten Algorithmen ist die Gewinnung von Gradienten aus der Formableitung eines Zielfunktionals, welche wir für die Generierung von Deformationen nutzen wollen, von zentraler Bedeutung. Das Skalarprodukt g^S bietet genau hierzu das geeignete Mittel, vgl. [27].

Definition 26 (Formgradient)**.** Seien die Voraussetzungen wie in 22 und 23. Sei $\mathcal{J}$ ein formdifferenzierbares Formfunktional und die zugehörige Formableitung $D\mathcal{J}$. Betrachte die zu $c \in B_e(S^1, \mathbb{R}^2)$ gehörige Form $\partial\Omega$, mit Tangentialraum T_cB_e. Dann heißt der Tangentialvektor $\operatorname{grad}\mathcal{J}(\Omega) \in T_cB_e$ *Formgradient von* $\mathcal{J}$ *in* Ω, falls seine Darstellung $\alpha_{\operatorname{grad}\mathcal{J}}$ als Skalarvektorfeld, gemäß 21,

$$g^S(\alpha_{\operatorname{grad}\mathcal{J}}, \alpha_V) = D\mathcal{J}(\Omega)[V] \quad \forall V \in C^1(\mathcal{D}, \mathbb{R}^2) \tag{3.1}$$

erfüllt, wobei $\alpha_V = \langle V_{|\partial\Omega}, n\rangle$ das Koeffizientenfeld des äußeren Normalenanteils von V auf dem Rand $\partial\Omega$ ist.

Die hier gemachte Definition ist als eine duale Repräsentation der Formableitung $D\mathcal{J}$ in dem Skalarprodukt g^S zu sehen. Damit folgen wir nicht der typischen Definition eines gewöhnlichen Gradienten, wie diese oft in der elementaren Analysis stattfindet, sondern betrachten diese in Anlehnung an den Riesz'schen Darstellungssatz. Der Aufwand bis zur Definition des Formgradienten macht auch deutlich, wie wesentlich die Art des Gradienten vom zugrunde liegenden Skalarprodukt abhängt, ganz im Gegenteil zur Formableitung, für welche kein zugrunde liegendes Skalarprodukt benötigt wird. Somit ergeben sich für die selbe Formableitung, je nachdem welche Metrik man für den Formraum verwendet, andere Gradienten und somit andere numerische Verfahren und Eigenschaften. Für eine Auswahl weiterer Alternativen, siehe [23], Kapitel 3.2.
Es fällt weiterhin auf, dass die hier geforderte Gleichung 3.1 nicht in dualer Paarung mit Skalarfeldern, welche in die Ableitung $D\mathcal{J}$ einfließen, sondern durch deren Koeffizientenfeldern der Normalenkomponente auf der Form berücksichtigt werden. Dies lässt sich mit dem Hadamard'schen Darstellungssatz

begründen:

$$\begin{aligned} D\mathcal{J}(\Omega)[V] &= \int_{\partial\Omega} f(s)\langle V(s), n(s)\rangle ds \\ &= \int_{\partial\Omega} f(s)\alpha_V(s)\langle n(s), n(s)\rangle ds \\ &= \int_{\partial\Omega} f(s)\alpha_V(s) ds \\ &= (f, \alpha_V)_{\mathcal{L}^2(\partial\Omega)} \end{aligned}$$

Damit wird klar, dass Vektorfelder $V \in C^1(\mathcal{D}, \mathbb{R}^2)$ mit gleichen Werten in äußerer Normalenrichtung n auf $\partial\Omega$ auch den selben Wert bei Ableitung erhalten, und somit lediglich die Koeffizientenfelder α_V auf der Form relevant sind. Die Identifikation mit diesen ist auch nötig, da wir das Skalarprodukt g^S für gerade diese Funktionen definiert haben, und nicht bezüglich der hierzu isomorphen echten Tangentialvektoren, welche Vektorfelder sind.
Mit diesem Zusammenhang lässt sich auch leicht der Bezug zu auf ganz $\mathcal{D}$ definierten Vektorfeldern V, und der Bilinearform a, welche g^S zu Grunde liegt, beschreiben. Wir formulieren die Zusammenhänge aus [27] für unsere Situation und Notation als Satz um, und geben einen Beweis.

Theorem 27 (Formgradienten und Deformationsfelder)**.** Seien die Voraussetzungen wie in Definition 26. Sei a eine koerzive, beschränkte, symmetrische Bilinearform, und g^S die zugehörige Steklov-Poincaré-Metrik. Weiterhin sei $\text{grad}\mathcal{J}(\Omega)$ der zu einer Form $\partial\Omega$ und Formfunktional $\mathcal{J}$ gehörige Formgradient. Dann existiert ein zu $\text{grad}\mathcal{J}(\Omega)$ gehöriges Vektorfeld $V_{\text{grad}\mathcal{J}} \in H_0^1(\mathcal{D}, \mathbb{R}^n)$, so dass gilt

$$g^S(\alpha_{\text{grad}\mathcal{J}}, \alpha_V) = D\mathcal{J}(\Omega)[V] = a(V_{\text{grad}\mathcal{J}}, V) \quad \forall V \in C^1(\mathcal{D}, \mathbb{R}^2) \tag{3.2}$$

und, mit Notation aus 22,

$$(\gamma_0 \circ V_{\text{grad}\mathcal{J}})^T n = \alpha_{\text{grad}\mathcal{J}}.$$

Beweis. Seien die Voraussetzungen wie oben. Nach der zuvor geführten Rechnung und mit der Definition des Formgradienten 26 gilt

$$g^S(\alpha_{\text{grad}\mathcal{J}}, \alpha_V) = D\mathcal{J}(\Omega)[V] = (f, \alpha_V)_{\mathcal{L}^2(\partial\Omega_2)},$$

wobei f aus der Randdarstellung von $D\mathcal{J}$ mittels Hadamard'schem Darstellungssatz stammt. Verwendet man die Definition 25 von g^S, so erhält man

$$(\alpha_V, (S^p)^{-1}\alpha_{\text{grad}\mathcal{J}})_{\mathcal{L}^2(\partial\Omega)} = \int_{\partial\Omega} \alpha_{\text{grad}\mathcal{J}}(S^p)^{-1}(\alpha_V)ds = \int_{\partial\Omega} f\alpha_V ds$$

für beliebige Vektorfelder $V \in C^1(\mathcal{D}, \mathbb{R}^2)$, da $(S^p)^{-1}$ symmetrisch in dualer Paarung bezüglich $(.,.)_{\mathcal{L}^2(\partial\Omega)}$ ist, oder äquivalent g^S symmetrisch ist. Nun folgt mit dem Fundamentallemma der Variationsrechnung und der Invertierbarkeit von S^p, dass

$$S^p(f) = \alpha_{\text{grad}\mathcal{J}}.$$

Es folgt mit der Definition von S^p, siehe 23, und mit der Koerzivität und Symmetrie von a, dass ein $V_{\text{grad}\mathcal{J}} \in H_0^1(\mathcal{D}, \mathbb{R}^2)$ existiert, so dass gilt

$$a(V_{\text{grad}\mathcal{J}}, V) = \int_{\partial\Omega} f\alpha_V ds \quad \forall V \in C^1(\mathcal{D}, \mathbb{R}^2),$$

woraus insgesamt die erste Gleichung 3.2 folgt. Die zweite Gleichung folgt direkt aus der Definition des projizierten Operators S^p , siehe 24, und des Neumann-Lösungsoperators E_N, denn

$$\alpha_{\text{grad}\mathcal{J}} = S^p(f) = n^T(\gamma_0 \circ E_N(f \cdot n)) = n^T(\gamma_0 \circ V_{\text{grad}\mathcal{J}}).$$

□

Mit diesem Resultat haben wir nun mehrere Möglichkeiten, aus gegebener Formableitung $D\mathcal{J}$ eine Deformation einer Form $\partial\Omega$ zu erzeugen. Zum einen besteht die Möglichkeit, auf dem Rand $\partial\Omega$ zu agieren, indem man ein Normalenvektorfeld $\alpha_{\text{grad}\mathcal{J}} \cdot n$ auf diesem mittels des Variationsproblems 3.2 auf Basis der linken Gleichheit erzeugt. Zu beachten ist, dass die so erzeugte Deformation lediglich auf dem Rand definiert ist, womit dann in der Anwendung das Gitter im Inneren und Äußeren nicht bewegt wird. Aus diesem Grund bezeichnen wir diese Art der Gewinnung von Deformationen als *Randformulierung*. Die andere Variante ist, eine Deformation auf dem gesamten Gebiet Ω zu gewinnen, indem man das Variationsproblem 3.2 auf Basis der rechten Gleichung mit der Bilinearform a löst. Im Gegensatz zur ersten Variante können hier potentiell alle Punkte des Gitters verschoben werden, trotz dessen, dass lediglich die Randpunkte zur Änderung des Zielfunktionals beitragen. Diese Variante bezeichnen wir als *Volumenformulierung*, angelehnt an die Arten der

Darstellung der Formableitung 2.1 aus dem vorigen Kapitel. Beide Varianten verwenden zur Erzeugung des Gradienten die Ableitungsinformation, welche physikalisch als Kraft interpretierbar ist. Wählt man als Beispiel für die Bilinearform a die lineare Elastizitätsgleichung, welche wir weiter unten definieren, so wird diese Interpretation noch deutlicher. Genau diese Bilinearform werden wir in der praktischen Implementierung wählen, um in Volumenformulierung Formgradienten zu erzeugen.
Wir erwähnen an dieser Stelle noch, dass die durch das Variationsproblem 3.2 erzeugte Deformationen des Randes h im Allgemeinen nicht in $C^1(\partial\Omega, \mathbb{R}^2)$ sind, siehe [27]. Je nach Wahl der Bilinearform a, welche den Operator S^p und somit das Skalarprodukt g^S erzeugen, sowie der rechten Seite der Gleichung und der Regularität des Gebietes auf der diese definiert sind, existiert lediglich eine Lösung in $H^{1/2}(\partial\Omega, \mathbb{R}^2)$. Aus diesem Grunde ist nach unserer Definition 21 grad$\mathcal{J}(\Omega)$ nicht immer Element des Tangentialraums T_cB_e, was Probleme theoretischer Natur macht. Unter anderem deshalb besteht an dieser Stelle der Bedarf eines allgemeineren Formraumes. Beispielsweise wäre eine Möglichkeit, Formen mit $H^{1/2}$-Rändern zuzulassen, was in [23], Kapitel 7, kurz erläutert wird. Hierbei entstehen allerdings neue Schwierigkeiten theoretischer Natur, welche noch nicht völlig erforscht sind. Für eine Einführung in diese Thematik und zugehörige Konzepte verweisen wir auf [22].
Da wir nun mit 3.2 verstanden haben, wie Formgradienten mit Hilfe von Metriken g^S und Bilinearformen a erzeugt werden können, möchten wir für dieses ein Beispiel angeben. Die Wahl einer solchen Bilinearform a, und damit die einer Metrik g^S, ist in der Tat nicht trivial, und mehrere Möglichkeiten mit unterschiedlichen Eigenschaften bestehen. Die Wahl hängt unter anderem davon ab, welches Symbol der Hesse-Operator des betrachteten Problems in der Lösung hat. Würde man eine Bilinearform wählen, sodass der dominierende Teil des Symbols des Hesse-Operators gleich ist, so vermuten wir, dass sich gitterunabhängige Konvergenz der Verfahren ergibt. Wir werden für die Gewinnung von Formgradienten im Nachfolgenden die lineare Elastizitätsgleichung verwenden, welche wir nun für unser Modellproblem einführen, wobei wir uns an [17] und [27] halten.

Definition 28 (Lineare Elastizitätsgleichung)**.** Betrachte das Gebiet $(0,1)^2 \subset \mathbb{R}^2$. Sei $f_{elas} : (0,1)^2 \to \mathbb{R}$ eine Funktion. Dann heißt die Differentialgleichung

$$\begin{aligned} \operatorname{div}(\sigma) &= f_{elas} \quad \text{in } (0,1)^2 \\ U &= 0 \qquad \text{auf } \partial([0,1]^2) \end{aligned} \tag{3.3}$$

mit,

$$\sigma := \lambda_{elas}\mathrm{Tr}(\epsilon)I + 2\mu_{elas}\epsilon$$
$$\epsilon := \frac{1}{2}(\nabla U + \nabla U^T),$$

lineare Elastizitätsgleichung in starker Formulierung, wobei $\mathrm{Tr}(\epsilon)$ die Spur der Matrix ϵ ist, und $\lambda_{elas} \in \mathbb{R}$, sowie $\mu_{elas} \in [0, \infty)$, die sogenannten *Lamé-Parameter* sind. σ wird als Dehnungs-, ϵ als Spannungstensor bezeichnet.

Die Lösung $U : \Omega \to \mathbb{R}^2$ in dieser Differentialgleichung lässt sich als physikalische Deformation auf dem Gebiet Ω auffassen, wobei f_{elas} als auf das Gebiet wirkende physikalische Kraft interpretiert werden kann.
Die Lamé-Parameter $\lambda_{elas}, \epsilon_{elas}$ besitzen hier keine physikalische Bedeutung, jedoch lassen haben diese Einfluss auf die Gitterdeformation, indem sie indirekt die Schrittweite steuern. Sie sind mit Hilfe des sogenannten *Young'schen Elastizitätsmoduls* E und der *Poissonzahl* ν durch

$$\lambda_{elas} = \frac{\nu E}{(1+\nu)(1-2\nu)},\ \epsilon_{elas} = \frac{E}{2(1+\nu)}$$

darstellbar. Das Elastizitätsmodul E lässt sich als Steifigkeit des Materials interpretieren. Die Poissonzahl ν gibt an, in welchem Verhältnis sich das einen Gitterpunkt umgebende Gitter ausdehnt, falls dieser Gitterpunkt in eine Richtung verschoben wird. Insgesamt lassen also die Lamé-Parameter durch ihre Wahl die Möglichkeit einer Art Schrittweitensteuerung zu. Im weiteren Verlauf dieser Arbeit wählen wir $\lambda_{elas} = 0$. Wir werden eine etwas erweiterte Variante der Linearen Elastizitätsgleichung zur Schrittsteuerung verwenden, welche *lokal variierende Lamé-Parameter* verwendet. Dabei wird das μ_{elas} als Koeffizientenfunktion $\mu_{elas} : (0,1)^2 \to [0, \infty)$ aufgefasst. Diese erzeugen wir für $\mu_{min}, \mu_{max} \in [0, \infty)$ durch das Lösen des Poisson-Problems

$$\begin{aligned} -\Delta\mu_{elas} &= 0 && \text{in } \mathcal{D} \\ \mu_{elas} &= \mu_{max} && \text{auf } \partial\Omega \\ \mu_{elas} &= \mu_{min} && \text{auf } \partial\mathcal{D}. \end{aligned} \tag{3.4}$$

Hierdurch lässt sich lokale Steifigkeit des Gitters gewährleisten, und somit der späteren Entartung von Gitterzellen etwas entgegensetzen. Je näher die Werte $\mu_{elas}(x)$ nahe 0, desto steifer bewegt sich der zugehörige Punkt x.
Möchte man nun mit Hilfe der linearen Elastizitätsgleichung auf Basis von Theorem 3.2 einen Formgradienten bestimmen, so besitzt man zwei Möglichkeiten. Zum einen lässt sich der Gradient mittels der Randformulierung der

Formableitung $D\mathcal{J}$ bilden. Hierzu wird eine weitere Dirichlet-Randbedinungung bei der lineare Elastizitätsgleichung 3.2, welche den Formgradienten bezüglich einer Sobolev-Metrik auf der Form $\partial\Omega$ repräsentiert, gebildet, und f_{elas} wird 0 gesetzt. Für Details hierzu, siehe [27]. Als andere Variante lässt sich die Volumenformulierung der Formableitung $D\mathcal{J}$ verwenden. Hierzu assembliert man bei der linearen Elastizitätsgleichung 3.3 die rechte Seite f_{elas} als den Volumenanteil $D\mathcal{J}_{target}(\Omega)[V]$, welchen wir in 2.11 angegeben haben . Die Perimeter-Regularisierung, welche nur von der Form $\partial\Omega$ abhängt, wird mit Hilfe einer zusätzlichen von-Neumann-Randbedingung der Form

$$\frac{\partial U}{\partial n} = f^{surf} \quad \text{auf } \partial\Omega$$

berücksichtigt. Wir werden in der Implementierung äquivalent hierzu die folgende schwache Formulierung der linearen Elastizitätsgleichung nutzen, wobei wir als rechte Seite die leichter auf mehrere Dimensionen verallgemeinerbaren Randformulierungen 2.12 und 2.14 verwenden. Die schwache Formulierung lautet dann

$$a_{elas}(U,V) := \int_{(0,1)^2} \sigma(U) : \epsilon(V)\, dx = D\mathcal{J}(\Omega)[V] \quad \forall V \in H_0^1((0,1)^2, \mathbb{R}^2). \tag{3.5}$$

3.5 ermöglicht nun bei relativ leichter Assemblierung die Berechnung eines Formgradienten durch lösen einer aus der linearen Elastizitätsgleichung erzeugten Variationsproblems, wobei wir die Rechtfertigung hierfür wie bei der Herleitung erwähnt in Theorem 27 finden. Wir werden für die praktische Implementierung 3.5 verwenden, wobei die Randformulierung prinzipiell als Alternative zur Verfügung stünde.

3.2 Quasi-Newton-Verfahren

Nun besitzen wir alle theoretischen Hintergründe aus dem Bereich der Formoptimierung, um für uns interessante, auf Ableitungen basierte Optimierungsverfahren einzuführen. Im Rahmen dieser Arbeit konzentrieren wir uns auf das sogenannte *Limited-Memory-Broyden-Fletcher-Goldfarb-Shanno-Verfahren (L-BFGS-Verfahren)*. Dieses Verfahren ist ein sogenanntes *Quasi-Newton-Verfahren*, welche wir im Folgenden nach [28] erläutern möchten. Bevor wir dies tun, geben wir noch Definitionen von Konvergenzgeschwindigkeiten numerischer Optimierungsmethoden an, vgl. [28], Appendix A2.

Definition 29 (Konvergenzraten)**.** Betrachte ein Minimierungsproblem, wobei x^* eine optimale Lösung ist. Sei $\{x_n\}_{n\in\mathbb{N}}$ eine Folge mit $x_n \to x^*$ in geeignetem Sinne. Dann heißt die Folge

i) *linear konvergent*, falls ein $c \in (0,1)$ existiert, mit

$$\frac{||x_{n+1} - x^*||}{||x_n - x^*||} \leq c \quad \text{für n hinreichend groß,} \tag{3.6}$$

ii) *superlinear konvergent*, falls gilt

$$\lim_{n\to\infty} \frac{||x_{n+1} - x^*||}{||x_n - x^*||} = 0, \tag{3.7}$$

iii) *quadratisch konvergent*, falls ein $c \in (0,1)$ existiert, mit

$$\frac{||x_{n+1} - x^*||}{||x_n - x^*||^2} \leq c \quad \text{für n hinreichend groß.} \tag{3.8}$$

Bevor wir den Fall der Quasi-Newton- und Newton-Verfahren in Formräumen diskutieren, beginnen wir mit der theoretisch gesicherten Betrachtung der Verfahren in endlichdimensionalen Vektorräumen. Das klassische Newton-Verfahren versucht, im Gegensatz zu einem Gradientenverfahren, auch Informationen zweiter Ordnung des Zielfunktionals in Form des Hesse-Operators $\mathrm{Hess}\mathcal{J}$ bei der Bildung eines Schrittes miteinzubeziehen. Verwendet man die übliche Notation im endlichdimensionalen Fall, bei der $x_k \in \mathbb{R}^n$ der Wert bei Schritt k ist, so berechnet sich der Schritt Δx über Lösen des Problems

$$\mathrm{Hess}\mathcal{J}(x_k)\Delta x = -\mathrm{grad}\mathcal{J}(x_k). \tag{3.9}$$

Verwendet man dieses Verfahren mit geeigneter Schrittweitensteuerung, so besitzt das volle Newton-Verfahren bei geeigneten Voraussetzungen quadratische Konvergenz, siehe [28]. Quasi-Newton-Verfahren ergeben sich nun aus dem Newton-Ansatz, indem man statt der Hesse-Matrix $\mathrm{Hess}\mathcal{J}(x_k)$ eine Approximation B_k für diese verwendet. Hierzu gibt es eine Vielzahl von Möglichkeiten. Genannt seien hier unter anderem die *Symmetric Rank 1 (SR1)* Updateformel und das *DFP-Verfahren*, siehe etwa [28]. Die BFGS-Update-Formeln für die Hesse-Matrix und ihre Inverse, falls existent, sind im endlichdimensionalen Fall wie folgt definiert, vgl. [28], 6.1.

Definition 30 (BFGS-Updates (endl. Dim.))**.** Sei B_k die Approximation der Hesse-Matrix Hess$\mathcal{J}(x_k)$, weiterhin seien

$$\begin{aligned} s_k &:= x_{k+1} - x_k \\ y_k &:= \text{grad}\mathcal{J}(x_{k+1}) - \text{grad}\mathcal{J}(x_k) \\ \rho_k &:= \frac{1}{y_k^T s_k}. \end{aligned}$$

Dann ist der *BFGS-Update* definiert durch

$$B_{k+1} := B_k - \frac{B_k(s_k s_k^T)B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}. \tag{3.10}$$

Das Inverse der Matrix B_{k+1} besitzt die Updateformel

$$B_{k+1}^{-1} := (I - \rho_k s_k y_k^T)B_k^{-1}(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T. \tag{3.11}$$

In der Definition ist nicht klar, was die natürliche Wahl für ein B_0, beziehungsweise B_0^{-1} ist. Hier gibt es mehrere Möglichkeiten, beispielsweise kann man eine mittels automatischem Differenzieren gewonnene Hesse-Approximation als Start B_0 wählen. Alternativ kann man die skalierte Identität αI mit $\alpha \in [0, \infty)$ wählen, um eine positiv definite Startmatrix zu erhalten.
Weiterhin sei bemerkt, dass man die DFP-Updates erhält, falls man die Inverse Updateformel für die eigentliche Hesse-Approximation B_k benutzt. Aufgrund dieser Tatsache nennt man die beiden Updates auch zueinander dual. Diese Formeln sind sogenannte *symmetric rank 2 Updates*, da sie eine symmetrische Approximation der Hesse-Matrix mit Updates vom Rang 2 erzeugen. Leitet man die BFGS-Updates auf abstraktem Wege her, so bietet sich die Möglichkeit an, zuerst B_{k+1}^{-1} zu definieren, und mit Hilfe der sogennanten Sherman– Morrison–Woodbury Formel den Update für B_{k+1} aus B_{k+1}^{-1} herzuleiten. Wir geben an dieser Stelle genau diese abstrakte definierende Eigenschaft der BFGS-Approximation B_{k+1}^{-1} an, vgl. [28], da wir in dieser eine mögliche Chance zur Verallgemeinerung einer Hesse-Approximation im Formraum sehen.

Theorem 31 (Definierende Eigenschaft des BFGS-Updates)**.** Es gelte die Notation aus 30. Betrachte das Optimierungsproblem

$$\begin{aligned} &\min_H ||H - B_k^{-1}||_{WFrob} \\ &\text{s.t. } H^T = H \text{ und } Hy_k = s_k, \end{aligned} \tag{3.12}$$

wobei B_k^{-1} die in 30 definierte Inverse des k'ten BFGS-Updates ist, und $||\cdot||_{WFrob}$ eine gewichtete Frobeniusnorm ist, siehe [28], 6.1. Dann ist der BFGS-Update der Inversen B_{k+1}^{-1} die eindeutige Lösung des Problems 3.12.

Wie man sieht, lässt sich die Inverse der BFGS-Approximation der Hesse-Matrix als Lösung eines beschränktes Optimierungsproblems für Matrizen definieren. Die einschränkenden Bedingungen entstehen hierbei auf natürliche Weise. Zum einen beschreibt die erste Nebenbedingung lediglich die Forderung einer symmetrischen Approximation, die zweite Nebenbedingung lässt sich, unter Verwendung der eingeführten Notation aus 30, äquivalent umformen zu

$$B_k(x_{k+1} - x_k) = \text{grad}\mathcal{J}(x_{k+1}) - \text{grad}\mathcal{J}(x_k), \tag{3.13}$$

was bekannt ist als *Sekantengleichung*. Um Existenz einer positiv definiten Inversen zu gewährleisten ist als Bedingung hinreichend, dass die Matrix B_k positiv definit ist. Setzt man dies voraus, so ergibt sich aus der Sekantengleichung als notwendige Bedingung

$$s_k^T y_k = s_k^T B_k s_k > 0. \tag{3.14}$$

Diese sogenannte *Curvature Condition* muss also notwendigerweise erfüllt sein, wenn man positive Definitheit von B_k erhofft. Andererseits führt diese auch dazu, dass ein BFGS-Update-Schritt, angewandt auf eine positiv definite Matrix B_k oder ihr Inverses B_k^{-1}, erneut eine positiv definite Matrix erzeugt, somit auch hinreichend für positive Definitheit ist.
Trivialerweise ist die bei konvexen Problemen die Curvature Condition immer erfüllt. Im nicht-konvexen Fall ist es trotzdem möglich mit Hilfe von Linesearch-Techniken, etwa der Wolfe oder starken Wolfe-Bedingungen, vgl. [28], die Curvature Condition zu erfüllen. Zu der Konvergenzgeschwindigkeit des BFGS-Verfahens in endlicher Dimension lässt sich sagen, dass unter Voraussetzungen, wie etwa Liptschitz-stetige Hesse-Operatoren, superlineare Konvergenz vorhanden ist, siehe [28], Theorem 6.6. Sind die genannten Bedingungen nicht erfüllt, so ist die Konvergenz des BFGS-Verfahrens im Allgemeinen, wie wir in unserer Implementierung zeigen werden, nicht gewährleistet.
Es fällt auf, dass man zur Berechnung eines BFGS-Updates die volle $n \times n$-Matrix B_k speichern muss, da diese in der Regel keine Dünnbesetztheit aufweisen, siehe [28], 7.2. Ob die in der genannten Quelle gemachte Argumentation, dass die Hesse-Matrix in der Regel nicht dünn besetzt ist, auch auf den Formfall übertragbar ist, zweifeln wir aufgrund des Hadamard'schen Darstellungssatzes an. Aus diesem Grund könnte problemabhängig ein Ausnutzen dieser Struktur, falls vorhanden, lohnenswert sein, was wir hier nicht vertiefen möchten. Da vor allem im Fall großer n, etwa bei feinen, höherdimensionalen Gittern in der Formoptimierung beträchtliche Kosten bei der Speicherung verursacht werden, bedient man sich der sogenannten *Limited Memory* Variante

der Verfahren. Hierzu wird eine Approximation an die BFGS-Matrix B_k erzeugt, indem man eine vorgegebene Anzahl l gespeicherten Schritte benutzt. Dies spart Speicherkapazität auf dem Rechner. Die formale Definition der approximativen Update-Schritte findet sich in [28], 7.(19), welche wir hier nicht angeben. Stattdessen verwenden wir die sogenannte *2-Schleifen-L-BFGS Rekursion*, welche wir weiter unten in 3.17 auf den Formfall übertragen können. Wir führen nun in analoger Weise zu dem klassischen Newton Verfahren von zuvor ein Lagrange-Newton-Verfahren in Falle der Formoptimierung ein, um anschließend mögliche Verallgemeinerungen des BFGS-Verfahrens auf den Formfall zu diskutieren. Da wir in Formräumen keine kanonische Vektorraumstruktur besitzen, benötigt man zur Verallgemeinerung des Newton-Verfahrens auf diese zusätzliche Techniken. Die Autoren von [24] liefern dafür einen Zugang unter Ausnutzung der Mannigfaltigkeitsstruktur, welche wir bereits zuvor in dieser Arbeit zur Konstruktion eines Formgradienten eingeführt haben. Hierzu konstruieren die Autoren auf Grundlage des Raums aller Formen eine neue Mannigfaltigkeit, welche im Wesentlichen ein Produkt aus Bündeln von den Formen $\partial\Omega$ abhängiger Hilberträume $H(\Omega)$ und dem Formraum B_e selber sind, siehe [24], unter Remark 1. Nun verwenden die Autoren Tangentialvektoren auf eben dieser Mannigfaltigkeit, um Richtungen zu gewinnen, mit derer Hilfe durch die Exponentialabbildung auf dem Formraum B_e ein Optimierungsschritt definiert werden kann. Formal lässt sich dies wie folgt notieren, wobei $\zeta = (h_y, h_\Omega, h_p)$ ein Tangentialvektor der oben genannten Mannigfaltigkeit von der Form analog zu 2.6 ist:

i) Löse das Newton-Problem

$$\mathrm{Hess}\mathcal{L}(\zeta_k)\Delta\zeta = -\mathrm{grad}\mathcal{L}(\zeta_k) \tag{3.15}$$

ii) Berechne den Schritt mit einer Schrittlänge α_k

$$\zeta_{k+1} := exp_{\zeta_k}(\alpha_k \Delta\zeta). \tag{3.16}$$

Wir haben das Lagrange-Newton-Verfahren im Formraum ausformuliert, um auf die offenen Schwierigkeiten und Fragestellungen der Übertragung der BFGS-Updates aus 30 aufmerksam zu machen. Da es sich bei 3.15 um eine Operatorgleichung handelt, anders als bei 3.9, welches ein LGS darstellt, ist nicht klar, welche Arten von Updates eine sinnvolle Verallgemeinerung der Updates 30 im endlichdimensionalen Fall für die Approximation des Hesse-Operators

im Formraum sind. Bei den Updates im endlichdimensionalen Fall wird stillschweigend eine euklidisches Skalarprodukt vorausgesetzt. Im Shape-Fall dürfte dieses durch eine geeignet gewählte Riemannsche Metrik, etwa g^S aus 25, ersetzt werden. Bei direkter Übertragung des BFGS-Updates liefert dies einen nicht komplett korrekten Update, was laut den Autoren dafür sorgt, dass keine asymptotische superlineare Konvergenz zu erwarten ist. Außerdem überträgt sich die Curvature Condition durch ersetzen des euklidischen Skalarprodukts mit g^S. Genau diesen Ansatz haben die Autoren von [26] in ihrem Paper unter Abschnitt 3 getan. Alternativ könnte man den Ansatz zur abstrakten Konstruktion 3.12 auf das Setting in Formräumen übertragen. Hierzu müsste man die Sekantengleichung mit Hilfe von Transporten in die richtigen Tangentialräume, welche weiter unten bei dem 2-Schleifen Algorithmus angegeben sind, formulieren. Fraglich ist die Bedingung der Symmetrie im Formraum-Fall, da der Hesse-Operator im Allgemeinen keine Symmetrie aufweist. Welche Bedingungen dann nötig wären, um Eindeutigkeit der Lösung zu erzwingen, ist offen. Weiterhin Bedarf es zur Formulierung des Problems einen geeigneten Abstand auf dem Raum der linearen Operatoren auf dem Tangentialbündel des Formraums. Wäre dies alles geschafft, so könnte man zumindest für abstrakte theoretische Untersuchungen an Quasi-Newton-Methoden die Existenz eines BFGS-Operators sichern. Wenn sich dieser Ansatz als fruchtbar erweisen sollte, so könnte sich dieser mit wahrscheinlich geringem Aufwand auf die gesamte Broyden-Klasse erweitern, da der DFP-Update als Lösung des selben Problems 3.12 erzeugt werden kann, wobei lediglich B_k^{-1} durch B_k ersetzt werden muss. Leider fehlt uns die Zeit für die Verfolgung dieses Ansatztes, zweitens ist der numerische Nutzen fragwürdig, da wahrscheinlich keine Verbesserung der Verfahren dabei entdeckt würde.
Aufgrund der genannten Schwierigkeiten werden wir uns vorerst mit der Übertragung der 2-Schleifen-L-BFGS Rekursion auf den Formfall begnügen. Hierzu haben die Autoren von [17] den Algorithmus aus [28], 7.4, unter Verwendung des Zusammenhangs 3.2 und der dort vorkommenden Metrik g^S, sowie der Bilinearform a, angepasst. Zu beachten ist, dass hier die Tangentialvektoren Elemente verschiedener Tangentialräumen sind, weshalb diese noch zuerst in die richtigen Räume transportiert werden müssen, was in unserem Fall der Tangentialraum des aktuellsten Gradienten ist. Für die differentialgeometrische Definition der Transporte $\mathcal{T}$, siehe etwa [6] oder [26] Abschnitt 3. Der klassische Algorithmus entsteht, wenn man statt der Metrik g^S im endlichdimensionalen Fall das euklidische Skalarprodukt verwendet, wobei keine Transporte von Tangentialvektoren anfallen. Es folgt die Definition nach [17], Kapitel

4.

Definition 32 (2-Schleifen-L-BFGS Rekursion im Formfall)**.** Seien die Voraussetzungen wie aus 3.2. Seien $\text{grad}\mathcal{J}(\Omega_i) \in T_{\partial\Omega_i}B_e$ Formgradienten der in den L-BFGS-Schritten erzeugten Formen $\partial\Omega_i$ mit ihren Darstellungen als Koeffizientenvektorfelder $\alpha_{\text{grad}\mathcal{J}_i} \in H^{1/2}(\partial\Omega_i)$ nach 21. Weiterhin seien $V_{\text{grad}\mathcal{J}_i} \in H_0^1(\mathcal{D}, \mathbb{R}^2)$ die zugehörigen Darstellungen auf der Hold-all-Domain $\mathcal{D}$ nach 3.2. Zudem seien $S_i \in H_0^1(\mathcal{D}, \mathbb{R}^2)$ Deformationsvektorfelder und $Y_i := V_{\text{grad}\mathcal{J}_{i+1}} - \mathcal{T}_{S_i} V_{\text{grad}\mathcal{J}_i} \in H_0^1(\mathcal{D}, \mathbb{R}^2)$ die entlang der Deformation S_i transportierte Differenz der Gradienten. Sei $l \in \mathbb{N}$ die Anzahl der gespeicherten Gradienten und Deformationen. Verwendet man Notation aus 22, dann ist die *2-Schleifen-L-BFGS Rekursion im Formfall* im Schritt j gegeben durch

$$
\begin{aligned}
&q \leftarrow V_{\text{grad}\mathcal{J}_j} \\
&\textbf{for } i = j-1, \dots, j-l \textbf{ do} \\
&S_i \leftarrow \mathcal{T}_{S_{j-1}} S_i \\
&Y_i \leftarrow \mathcal{T}_{S_{j-1}} Y_i \\
&\rho_i \leftarrow g^S((\gamma_0 Y_i)^T n, (\gamma_0 S_i)^T n)^{-1} = a(Y_j, S_j)^{-1} \\
&\alpha_i \leftarrow \rho_i g^S((\gamma_0 S_i)^T n, (\gamma_0 q)^T n) = \rho_i a(S_i, q) \\
&q \leftarrow q - \alpha_i Y_i \\
&\textbf{end for} \\
&q \leftarrow \frac{g^S((\gamma_0 Y_{j-1})^T n, (\gamma_0 S_{j-1})^T n)}{g^S((\gamma_0 Y_{j-1})^T n, (\gamma_0 Y_{j-1})^T n)} q = \frac{a(Y_{j-1}, S_{j-1})}{a(Y_{j-1}, Y_{j-1})} q \\
&\textbf{for } i = j-l, \dots, j-1 \textbf{ do} \\
&\beta_i \leftarrow \rho_i g^S((\gamma_0 Y_i)^T n, (\gamma_0 q)^T n) = \rho_i a(Y_i, q) \\
&q \leftarrow q + (\alpha_i - \beta_i) S_i \\
&\textbf{end for,} \qquad\qquad S_j \leftarrow q
\end{aligned}
\tag{3.17}
$$

wobei $\mathcal{T}_{S_{j-1}}$ der Transport in den entsprechenden Tangentialraum des aktuellen Gitters ist, welches durch Deformation S_{j-1} entstanden ist. Dann lässt sich das erzeugte Vektorfeld als Deformation $S_j := q = \tilde{B}_j^{-1} V_{\text{grad}\mathcal{J}_j}$ verwenden, wobei $\tilde{B}_j$ die L-BFGS-Approximation des Hesse-Operators in Schritt j ist. Dieser Algorithmus ist ausgelegt für Maximierungsprobleme, möchte man ein Minimierungsproblem lösen, wie wir es tun, so muss man entweder $q \leftarrow -V_{\text{grad}\mathcal{J}_j}$ setzen, oder wegen Linearität äquivalent
$S_j := -q$ setzen.

Die Transporte und das Speichern der entsprechend transportierten Vektorfelder bewirkt, dass sich die Vektoren der vorherigen Schritt im j'ten Schritt im selben Tangentialraum befinden, und somit lediglich den Transport in den aktuellen benötigen. Wir möchten den Leser darauf aufmerksam machen, dass mit einfach Indizierten $\alpha_i \in \mathbb{R}$ Skalare und mit $\alpha_{\text{grad}\mathcal{J}_i} \in H^{1/2}(\partial\Omega_i)$ Skalarvektorfelddarstellungen der Gradienten gemeint sind, um Verwechslung zu vermeiden. In der praktischen Realisierung werden wir keine exakten Transporte verwenden, da diese den Rechen- und Implementierungsaufwand deutlich steigern, mehr hierzu im nächsten Kapitel. Stattdessen werden wir die Tangentialvektoren als Vektorfelder nicht verschieben, sondern diese lediglich als in den passenden Tangentialraum gebracht erachten, was einer Vereinfachung wie in der oben genannten Quelle entspricht. Somit fällt für die Implementierung kein Transport an, und wir verwenden die Vektorfelder wie diese erzeugt wurden.
Mit Abschluss dieses Kapitels haben wir alle theoretischen Aspekte für die praktische Behandlung der in den nächsten Kapiteln folgenden Implementierung und deren Resultate gelegt. Wir möchten nun die Umsetzung von Algorithmus 3.17 erläutern.

4 Implementierung in Python mit FEniCS

In diesem Abschnitt möchten wir die Implementierung[1] des Algorithmus 3.17 in Python 3.5 mit Hilfe des Moduls FEniCS vorstellen. Im Folgenden werden wir die in den Dateien enthaltenen Kommentare nicht oder nicht in voller Länge in den Codeausschnitten aufführen, da wir Redundanz bei den Erklärungen vermeiden möchten. Selbstverständlich sind in den Quellcodes selber ausführliche Kommentierungen vorhanden, weshalb wir gerne auf diese verweisen. Die Implementierung in Python besteht zum Hauptteil aus den beiden Dateien

shape_main.py shape_bib.py.

Die Datei *shape_main.py* enthält hierbei den zusammenhängenden Hauptcode. Die Datei *shape_bib.py* ist eine Bibliothek, in welcher Funktionen zum Umgang mit Gittern, Berechnungen auf Formen, Löser für PDEs und der 2-Schleifen-L-BFGS-Algorithmus mit den damit verbundenen Objekten gebündelt sind. Beide Dateien, mit samt den von uns im nächsten Kapitel präsentierten Ergebnissen, sind in der Abgabe mit enthalten. Zusätzlich ist eine

[1] Die Funktionsfähigkeit des Programms ist ausschließlich unter der Version Python 3.5, sowie den zugehörigen *FEniCS*-Paketen gewährleistet.

Testimplementierung der Objekte für das BFGS-Verfahren zur Lösung eines endlichdimensionalen Optimierungsproblems beigefügt, der zeigt, dass die Objekte alle das korrekte Verhalten haben. Die Berechnungen auf den Formen und das Lösen der PDEs wird mittels FEniCS geschehen. FEniCS[2] ist eine frei zugängliches Programmpaket, welches ermöglicht, partielle Differentialgleichungen mit relativ geringem Aufwand zu lösen. Dabei bedient sich FEniCS der sogenannten *Unified Form Language* (UFL), was die Grundlage zur Implementierung der PDE's in schwacher Formulierung darstellt, mehr hierzu bei [8]. Bevor wir uns der Lösung von PDE's und der Implementierung des L-BFGS-Algorithmus zuwenden, müssen wir zunächst klären, wie wir die notwendigen Gitter erzeugen und mit diesen umgehen.

4.1 Gittererzeugung

Gitterdateien erzeugen wir mit Hilfe des offen zugänglichen Programms Gmsh 3.0.6[3]. Hierbei muss zunächst eine .geo Datei geschrieben werden. Wir zeigen dies am Beispiel eines kleinen Kreises, welcher für unsere. Zunächst setzen wir die für unser Gitter relevanten Punkte in ein 3-dimensionales Koordinatensystem

```
Point(1) = {0.0, 0.0, 0.0, 1.0};
Point(2) = {1.0, 0.0, 0.0, 1.0};
Point(3) = {0.0, 1.0, 0.0, 1.0};
Point(4) = {1.0, 1.0, 0.0, 1.0};
Point(5) = {0.5, 0.35, 0.0, 1.0};
Point(6) = {0.5, 0.5, 0.0, 1.0};
Point(7) = {0.5, 0.65, 0.0, 1.0};
```

Hierbei beschreiben die ersten 3 Einträge des Tupels die x-, y- und z-Koordinaten der Punkte, der vierte Eintrag gibt die sogenannte *characteristic length* der Punkte an, was lediglich die Elementgröße des Punktes ist. Punkte 1 bis 4 werden dazu dienen das Einheitsquadrat im $\mathbb{R}^2$ zu definieren, Punkte 5 bis 7 werden einen Kreis mit Mittelpunkt $(0.5, 0.5)$ definieren. Dies geschieht mittels der Befehle

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

```

[2]Weitere Informationen sind unter https://fenicsproject.org/ zu finden.
[3]Weitere Informationen unter http://gmsh.info/.

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

Da diese Befehle lediglich Linien und Halbkreise aus den eingegebenen Punkten definieren, ist es nötig mittels eines Loop-Befehls diese zu einer gemeinsamen Form zu verbinden.

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

Um innere und äußere Gebiete, welche durch Abgrenzung mittels des Kreises definiert sind, zu markieren, setzen wir diese als Plane Line fest. Die erste Zahl gibt jeweils die Nummer des Line Loops des äußeren Randes, die zweite die des inneren Randes an.

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

Weil wir später in der Implementierung auf die Ränder bzw. Formen zugreifen möchten, ist es abschließend noch nötig diese als sogenannte Physical Lines und Physical Surfaces zu definieren.

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

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

Die so geschriebene .geo-Datei wird nun mit Hilfe des Programms Gmsh in der Konsole mit dem Kommando

gmsh mesh_smallercircle.geo -2 -clscale 0.1

in eine .msh Datei konvertiert. Dabei ist mesh_smallercircle.geo der Name der gespeicherten .geo-Datei, -2 die Dimension des erzeugten Gitters und 0.1 die Feinheit des Gitters. Um diese Datei für FEniCS nutzbar zu machen, konvertieren wir diese mit Hilfe des Dolfin-Befehls

dolfin-convert mesh_smallercircle.msh mesh_smallercircle.xml

wobei der erste Eingabewert der Name der .msh-Datei ist, und der zweite der Name der erzeugten .xml-Datei. Hierbei werden außerdem neben dem bloßen

Mesh auch eine facet_region.xml-Datei erstellt, mit welcher man die Ränder initialisieren kann, sowie eine physical_region.xml-Datei, welche zur Initialisierung der Gebiete des Inneren und Äußeren der Form dient. Beispielgitter finden sich im nächsten Kapitel über die Resultate, etwa in 2.

4.2 Die Bibliothek shape_bib

4.2.1 Die MeshData-Klasse

Nun besitzen wir die nötigen Gitterdateien, um auf diesen Formoptimierung zu betreiben. Wir stellen kurz vor, mit welchen Objekten und Funktionen wir auf diesen operieren. Eines der beiden zentralen Klassen des Optimierungsprogramms ist die sogenannte *MeshData-Klasse*.

```
class MeshData:

    # Objekt mit allen Daten des Gitters
    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 mit Traeger nicht
        # am inneren Rand
        self.indNotIntBoundary = ind
```

In einem Objekt dieser Klasse werden sowohl das Gitter, als auch die Gebiete und Ränder bzw. Formen gespeichert. Weiterhin benötigen wir für spätere folgende Berechnungen auch die Indizes der Knotenpunkte (engl. *Vertices*), welche keinen Träger am inneren Rand haben, gespeichert. Die Initialisierung erfolgt mit der von uns implementierten Funktion

load_mesh(Name),

wobei Name der Name der Mesh-Datei ohne .xml-Endung ist. Die subdomains und boundaries werden jeweils als ein sogenannte MeshFunction initialisiert. Dies sind Objekte einer in FEniCS implementierten Klasse, welche als Array im i-ten Eintrag die Nummer der subdomain bzw. der boundary zurückgeben,

welche den Nummern der Physical Surface bzw. Physical Line in der .geo-Datei entsprechen. Diese Initialisierungen geschehen über die Befehle

```
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)
```

wobei Mesh als Eingabe den Pfad zur .xml-Datei enthält. Die Meshfunktionen erhalten neben dem mesh-Objekt den Typ der Funktion, in diesem Fall size_t, und die Pfade zu den jeweiligen Dateien _physical_region.xml und _facet_region.xml. Es bleibt noch, die Indexliste der Indizes mit Träger nicht am Inneren Rand zu initialisieren. Um diese Indexliste zu erzeugen haben wir die Funktion

__get_index_not_interior_boundary(mesh, subdomains, boundaries, interior = True)

implementiert. Als Input erhält sie die oben gezeigten Objekte, falls interior = True eingestellt ist, so gibt die Funktion die Liste mit Indizes ohne Träger am inneren Rand wieder. Wir möchten an dieser Stelle anmerken, dass Indizes auch mehrfach vorkommen, was für unser Programm kein Problem darstellt und bei Bedarf verbessert werden kann. Ist der Parameter interior = False, so gibt die Funktion eine Liste mit den Indizes der Knotenpunkte genau des inneren Randes wieder. Dies spart uns die Implementierung einer weiteren Funktion. Das erzeugen der Liste basiert auf Iterationen durch Facetten des Randes, deren Knoten und den benachbarten Knoten. Diese aufwändige Iteration ist nötig, da die Indizierung der Facetten in der Meshfunktion des Randes in FEniCS nicht mit den Indizes der Mesh's übereinstimmen. Für die genaue Implementierung verweisen wir auf den von uns beigefügten Code.

4.2.2 Berechnungen in der shape_bib

Der Hauptanteil der Berechnungen besteht in dem Lösen von partiellen Differentialgleichungen. Die Gleichungen, welche wir für die Formoptimierung lösen müssen, haben wir in den vorigen Kapiteln eingeführt. Dabei handelt es sich um die Poisson-Zustandsgleichung mit Interface Condition aus unserem Modellproblem 2.7, die zugehörige adjungierte Gleichung 2.10, und die lineare Elastizitätsgleichung 3.3, welche uns einen Formgradienten liefern wird. Wir

möchten exemplarisch an der implementierten Funktion zur Lösung der linearen Elastizitätsgleichung zeigen, wie dies in FEniCS praktisch passiert. Für die beiden anderen Gleichungen wird analog vorgegangen, hierbei verweisen wir auf unseren Quellcode. Die Lösung der Gleichungen wird über die Funktionen

```
solve_state(meshData, fValues)
solve_adjoint(meshData, y, z)
solve_linelas(meshData, p, y, z, fValues,
              mu_elas, nu, zeroed = True)
```

zurückgegeben, wobei jede Funktion ein Objekt der MeshData-Klasse erhält, sowie die nötigen Funktionen aus den oben genannten Gleichungen, welche als sogenannte *FEniCS-Funktionen* initialisiert sind. Der Löser der linearen Elastizitätsgleichung benötigt weiterhin die in 3.4 eingeführten lokal-variierenden Lamé-Parameter, sowie den Parameter der Perimeter-Regularisierung nu. Zusätzlich gibt dieser die Norm des Objekts, welches die Formableitung als Operator auf dem Raum der Testfunktionen repräsentiert, als zweiten Return wieder, mehr hierzu im weiteren Verlauf bei 4.1. Die Einstellung zeroed = True bewirkt, dass bei Lösen der Gleichung die Werte der Punkte ohne Träger am inneren Rand auf 0 gesetzt werden, was eine Instabilität des Verfahrens vermeidet. Wir vermuten, dass es sich bei den Instabilitäten um Rundungs- und/ oder Diskretisierungsfehler handelt, siehe [27], Abschnitt 5. Die genaue Ursache der Fehler ist jedoch nicht sicher geklärt.
Wie oben schon erwähnt, sind die Funktionen p, y, z FEniCS-Funktionen. Skalarwertige FEniCS-Funktionen f auf dem Gitter meshData.mesh werden beispielsweise mit

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

initialisiert. Um die lineare Elastizitätsgleichung zu lösen, müssen wir zunächst den zugehörigen Funktionenraum angeben, was durch

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

geschieht. Da die Lösung eine vektorwertige Funktion ist, ist es für FEniCS notwendig die Dimension explizit anzugeben. Die Parameter "P" und 1 geben an, dass die Werte zwischen den Gitterpunkten mittels einer Polynominterpolation vom Grad 1 erzeugt werden. Hier sind weitere Möglichkeiten zur Interpolation gegeben, siehe [7]. Um die Gleichung aufzustellen, müssen wir Randwerte festlegen. Für die Dirichlet-Nullrandwerte geschieht dies über den Befehl

```
u_out = Constant((0.0, 0.0))
bcs = [DirichletBC(V, u_out,
                   meshData.boundaries, i) for i in range (1, 5) ]
```

wobei i über die Nummern der äußeren Ränder läuft. FEniCS unterscheidet beim Lösen von Differentialgleichungen zwischen Testfunktionen und der Lösungsfunktion. Der Unterschied zu normalen Funktionen ist, dass mit diesen bei dem Assemblieren der schwachen Formulierung die Terme, etwa die Bilinearform a weiter unten, nicht ausgewertet werden können, im Unterschied zu regulären FEniCS-Funktionen. Wir initialisieren diese mittels

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

wobei U die Lösung, also das Gradientenvektorfeld in Domaindarstellung, und v stellvertretend für die zum Raum V gehörenden Testfunktionen steht. Nun wird die linke und rechte Seite der Gleichung aufgestellt:

```
LHS     = bilin_a(meshData, U, v, mu_elas)
F_elas = shape_deriv(meshData, p, y, z, fValues, nu, v)
```

Hierbei ist bilin_a die aus 3.5 bekannte Bilinearform, und shape_deriv die Formableitung $D\mathcal{J}(\Omega)[V]$ als Summe von den Darstellungen 2.12 und 2.14. Beides wird in der für FEniCS typischen Weise assembliert, wobei wir dies exemplarisch an der Bilinearform bilin_a aufzeigen:

```
def bilin_a(meshData, U, V, mu_elas):

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

    epsilon_V = sym(nabla_grad(V))
    sigma_U   = 2.0*mu_elas*sym(nabla_grad(U))

    a     = inner(sigma_U, epsilon_V)*dx('everywhere')
    value = assemble(a)

    return value
```

Input sind ein Objekt der MeshData-Klasse, sowie zwei FEniCS Funktionen U, V und die lokal-variierenden Lamé-Parameter mu_elas. Hier kommt die Stärke von FEniCS zur Geltung, nämlich die Verwendung der eingangs erwähnten *Unified Form Language*. Die hier initialisierten Objekte sind exakt die Objekte, welche in der schwachen Formulierung der Gleichung 3.3 auftauchen, was der mathematischen Schreibweise sehr nahe steht, und somit die Lesbarkeit und Übersicht deutlich erhöht. Es ist lediglich notwendig die Objekte abschließend zu assemblieren um einen Wert zu erhalten. Dies geschieht mit dem Befehl assemble. Genau auf selbige Weise wird die Formableitung shape_deriv aufgebaut, weshalb wir hier auf den Quellcode verweisen. Da die Angabe der pro-

grammiertechnischen Details der Objekte der UFL den Rahmen dieser Arbeit sprengen würde, verweisen wir für den interessierten Leser auf [7] und [8].
Es bedarf nur noch dem Initialisieren der Randbedingungen für die assemblierten Objekte, bevor wir die lineare Elastizitätsgleichung lösen. Zuvor bauen wir noch die Option ein, die Werte, welche nicht am Träger des inneren Randes sind, auf Null zu setzen.

```
if(zeroed): F_elas[meshData.indNotIntBoundary] = 0.0

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

Das lösen der nun aufgebauten Gleichung erfolgt mit dem Befehl.

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

# Rueckgabe des Deformationsvektorfeldes U und der
    Abbruchbedingung
return U, nrm_f_elas
```

Zu beachten ist, dass wir U als .vector() Objekt übergeben. Diese Objekte lassen Arithmetik, wie beispielsweise das Initialisieren einzelner Werte an Knoten des Gitters für die FEniCS-Funktion, zu. Dies werden wir bei der Implementierung des BFGS-Schrittes maßgeblich verwenden. Das Objekt F_elas ist bereits von diesem Typ, da wir nicht mit einer bestimmten Richtung, sondern mit einer TestFunction initialisiert haben, weshalb wir hier keinen Befehl benötigen. Das so entstandene Objekt ist also keine skalare Größe, sondern lässt sich als

$$D\mathcal{J}(\Omega)[\varphi_i] \hat{=} \text{F\_elas.get\_local()[i]} \tag{4.1}$$

interpretieren, wobei φ_i polynomielle Basisfunktionen vom Grad 1 auf dem Gitter sind, welches Ω repräsentiert. Auf diese Weise lässt sich $D\mathcal{J}(\Omega)[\cdot]$ über F_elas als skalarwertige Funktion auf dem initialisierten Gitter auffassen, wobei die jeweiligen Werte des Knoten mit Index i genau den Ableitungen $D\mathcal{J}(\Omega)[\varphi_i]$ entsprechen. Dies ermöglicht es uns, die $\mathcal{L}^2$-Norm des so zur Formableitung assoziierten Objekts zu bilden, welche genau der zweite Return nrm_f_elas ist. Die Größe dieser Norm wird bei uns als Ausstiegskriterium sowohl bei dem Gradienten-, als auch bei dem L-BFGS-Verfahren Verwendung finden. Wir warnen den Leser an dieser Stelle, dass die obige Indizierung der Gitterpunkte i und somit der Basisfunktionen φ_i im Allgemeinen nicht mit der Indizierung der Werte der FEniCS-Funktionen auf dem Gitter, welche *Degrees of freedom*

(DOF) genannt werden, und somit nicht mit F_elas.get_local()[i] übereinstimmen. Dennoch lässt sich eine entsprechende Bijektion finden, welche in Dolfin mit dem Befehl Dolfin.vertex_to_dof_map(V) erzeugt wird. Diese wird bei uns zum berechnen eines nötigen Formabstandes verwendet, auf welchen wir nun zu sprechen kommen.
Die Distanz zweier Formen werden wir mit der Funktion

$$d_{shp}(\partial\Omega_1, \partial\Omega_2) := \int_{\partial\Omega_2} \min_{y \in \partial\Omega_1} ||x - y|| dx \tag{4.2}$$

messen. Diese Distanzfunktion wird bei uns lediglich als Ersatz für den Abstand zweier Formen im Formraum verwendet, welchen man mittels Geodätischer definieren kann, und ist eine Abwandlung des Hadamard-Abstandes. Um Geodätische zu bestimmen wäre es nötig eine weitere Differentialgleichung zu lösen, was wir vom Aufwand für nicht vertretbar halten, obwohl dies die korrektere Variante zur Abstandsmessung wäre. Wir bemerken außerdem, dass die oben definierte Abstandfunktion d_{shp} nicht symmetrisch ist, was sich auch anhand der von uns implementierten Beispielformen leicht vorführen lässt. Die Auswertung dieser Distanzfunktion passiert über die von uns implementierte Funktion

```
1 mesh_distance(mesh1, mesh2)
```

wobei mesh1, mesh2 Objekte der MeshData-Klasse sein müssen. Die eigentliche Berechnung der Minima geschieht über Iteration aller sich auf den jeweiligen Boundaries befindlichen Punkte, deren Vertex-Indizes wir aus den Facettenindizes beispielsweise mittels

```
___get_index_not_interior_boundary(mesh2.mesh, mesh2.subdomains,
                                   mesh2.boundaries, interior = False)
```

erhalten. Wie bei der linearen Elastizitätsgleichung erwähnt, lässt sich die Liste der Minima der einzelnen Punkte x nicht ohne weiteres als FEniCS-Funktion initialisieren, da der angesprochene Unterschied der Indizierung der Gitterpunkte und DOFs Probleme bereitet. An dieser Stelle kommt die vertex_to_dof_map zum tragen. Abschließend findet eine zu 4.2.2 ähnliche Assemblierung des Integrals statt, welche uns den Abstandwert als Return liefert.

4.2.3 Die BFGS-Memory-Klasse

Die von uns bisher eingeführten Objekte würde schon ausreichen, um ein Verfahren auf Basis des Gradientenabstiegs zu programmieren. Wir möchten

jedoch als Ziel dieser Arbeit einen Schritt weiter gehen, und den L-BFGS-Algorithmus implementieren. Das Hauptproblem bei dem Verfahren besteht darin, sich eine Möglichkeit schaffen, mit dessen Hilfe man die Gradienten- und Deformationsfelder bequem speichern und updaten kann, da wir ja nur eine *limited memory* besitzen. Hierzu entwerfen wir die zweite wichtige Klasse in unserem Programm, die sogenannte bfgs_memory-Klasse. Diese ist wie folgt aufgebaut, wobei wir im Programm selber noch Initialisierungsfehler mit Warnungen an künftige weitere Verwender eingebaut haben, siehe den Quellcode:

```
1  class bfgs_memory:
2
3      def __init__(self, gradient, deformation, length, step_nr):
4
5          # Liste von Gradientenvektorfeldern
6          self.gradient = gradient
7
8          # Liste von Deformationsvektorfeldern
9          self.deformation = deformation
10
11         # Anzahl der gespeicherten letzten Schritte
12         self.length = length
13
14         # Anzahl der bereits ausgefuehrten l-BFGS-Schritte
15         self.step_nr = step_nr
```

Diese Klasse besteht aus zwei Listen von Arrays, gradient und deformation, und zwei Integer-Werten length und step_nr. length ist der Parameter, welche die Anzahl der gespeicherten vorherigen Schritte im L-BFGS-Verfahren angibt, und somit die Länge der Liste der Arrays bestimmt. step_nr ist ein Counter, welcher die Schritte des L-BFGS-Verfahrens zählt.
Die Gradienten und Deformationen werden als Arrays, welche jeweils die DOF-Werte der FEniCS-Funktionen der Gradienten- und Deformationsvektorfelder enthalten, dem Alter her aufsteigend gespeichert. Das bedeutet beispielsweise, dass mit

bfgs_memory.gradient[1]

die Liste der DOF-Werte des vorletzten Gradientenvektorfeldes abgerufen werden. Die Speicherung als Array, und nicht als FEniCS-Funktion, ist nötig, da wir damit den Transport der Vektorfelder umgehen. FEniCS-Funktionen sind unweigerlich an das Gitter, auf dem sie initialisiert wurden, gebunden, und eine Änderung des zugrunde liegenden Gitters macht die Funktionen unbrauchbar. Da wir die Indizierung der Knoten, und damit die der DOFs, nicht durch Deformationen verändern, entkoppeln wir die Funktionswerte von dem Gitter,

indem wir ausschließlich diese nach der DOF Indizierung speichern. Damit erreichen wir den vereinfachten Transport nach [26], Abschnitt 4. Anschließend lässt sich bei Bedarf mit diesen Daten eine neue FEniCS-Funktion auf einem neuen Gitter initialisieren.

```
def initialize_grad(self, meshData, i):

    if isinstance(meshData, MeshData): pass
    else: raise SystemExit("initialize_grad benoetigt Objekt der
   MeshData-                    Klasse als Input!")

    V = VectorFunctionSpace(meshData.mesh, "P", 1, dim=2)
    f = Function(V)
    f.vector()[:] = self.gradient[i]
    return f
```

wobei es wichtig ist, ein Objekt der MeshData-Klasse zu übergeben. Es ist wichtig zu beachten, dass die Initialisierung von Werten in die FEniCS-Funktion ausschließlich durch einen kompletten Slice-Befehl auf allen Gitterpunkten erfolgen sollte, da FEniCS sonst eine automatische Konvertierung der Funktion durchführt und diese für einige spätere Berechnungen unbrauchbar macht. Da wie angesprochen die Indizierung invariant bei Verschiebung ist, ist dies kein Problem.
Um die Memory zu updaten, haben wir eine Update-Funktion implementiert, welche den ältesten Wert, sobald die Anzahl length an gespeicherten Einträgen überschritten wird, löscht, und alle anderen dem Index nach um 1 aufrückt.

```
def update_grad(self, upd_grad):

    for i in range(self.length-1):
        self.gradient[-(i+1)] = self.gradient[-(i+2)]

    self.gradient[0] = upd_grad
```

Wir wollen anmerken, dass sich diese Klasse natürlich auch direkt für die Implementierung von anderen Limited-Memory-Verfahren weiterverwenden ließe. Mit Hilfe dieser Klasse lässt sich nun ein L-BFGS-Schritt nach dem 2-Schleifen-Algorithmus 3.17 implementieren. Dies haben wir mit der Funktion

bfgs_step(meshData, memory, mu_elas, q_target)

getan. Diese Funktion benötigt als Input zum einen ein Objekt der MeshData-Klasse, auf der die FEniCS-Funktionen initialisiert, und die Bilinearform bilin_a ausgewertet werden. Für die Bilinearform werden zudem die Lamé-Parameter mu_elas benötigt. Weiterhin wird eine bfgs_memory verlangt, mit

deren Daten die genannten Funktionen erzeugt werden. Die von uns in vorigen Beispielen erläuterte Arithmetik mit FEniCS-Funktionen kommt hierbei zum tragen. Da dies zu langen Codesequenzen führt, verweisen wir für die genaue Implementierung auf den beiliegenden Quellcode. q_target ist hier ein Array mit den DOF-Werten einer FEniCS-Funktion, in welcher der in dem Schritt berechnete approximierte Hesse-Operator ausgewertet wird. So gibt die Funktion bfgs_step eine vektorwertige FEniCS-Funktion zurück, welche als Deformationsvektorfeld dienen kann, und mit

$$-B_k^{-1} q_{target} \hat{=} \text{bfgs\_step(meshData, memory, mu\_elas, q\_target)}$$

identifiziert werden kann, falls $k > 0$. Im Falle, dass die Memory noch leer ist, und kein Schritt zuvor durchgeführt wurde, gibt die Funktion das negative Eingabevektorfeld $-q_{target}$ wieder. Wir möchten den Leser darauf aufmerksam machen, dass die Position der in der Memory gespeicherten Gradienten und Deformationen für die korrekte Berechnung des Schrittes essentiell sind. Befindet man sich in Schritt k, so muss der Eintrag memory.gradient[0] der k'te Gradient sein, und memory.deformation[0] die zuvor im Schritt $k-1$ berechnete Deformation S_{k-1}. D.h. die Memory muss auf dem aktuellst möglichen Stand sein. Jetzt besitzen wir das gesamte Rüstzeug, um in der Hauptdatei die Bausteine miteinander zu vernetzen.

4.3 Das Hauptprogramm shape_main

4.3.1 Der Hauptalgorithmus und Einstellungen

Die von uns eingeführten Funktionen und Objekte lassen sich nun auf bequeme Weise miteinander in Verbindung bringen. Die Hauptdatei führt, je nach Einstellung der Parameter, eine Formoptimierung mit in der Datei verstellbarem Verfahren und Gittern durch. Zunächst zur Auswahl der Gitter; diese müssen in der Konsole bei Aufruf des Programms als integer-Parameter mit dem Befehl

```
python3.5 shape_main.py -m 9
```

übergeben werden, wobei wir hier das Beispielgitterpaar 9, welches für die Gitterkombination

```
("mesh_fine_smallercircle", "mesh_fine_broken_donut")
```

steht, verwendet wird. Eine Liste der Gitter kann mit dem Befehl

python3.5 shape_main.py -h

angezeigt werden. Das Startgitter wäre in diesem Fall das eines fein aufgelösten kleinen Kreises, der zweite Eintrag das Zielgitter eines fein aufgelösten, nierenartigen Form, zu sehen in Abbildung 2.
Wir haben bei der Wahl des Gitters außerdem die Option eingebaut, bei der man als Startgitter mit einer in jedem Punkt am Rand unabhängig normalverteilt gestörten Version des ersten Eintrags der Gitterpaare startet, und als Zielgitter das ungestörte Startgitter verwendet. Diese Option lässt sich mit den Parametern Pertubation und sigma steuern, wobei der erste Parameter diesen Optimierungsfall bei Pertubation = True einschaltet, und der zweite Parameter die Standardabweichung der Normalverteilung angibt. Wir wollen darauf hinweisen, dass bei großem sigma aufgrund der Unabhängigkeit der Verteilungen auch entartete Gitter entstehen können, so dass das Verfahren direkt abbricht. Abgebrochen wird die Optimierung, sobald der bei 4.1 eingeführte Wert der Norm nrm_f_elas die einstellbare Toleranz tol_shopt unterschreitet.
Als Parameter des Ausgangsproblems lassen sich außerdem noch die Werte $f_1, f_2 \in \mathbb{R}$ der Zustandsgleichung in 2.7 innerhalb und außerhalb der Form, f_1 und f_2, einstellen, sowie auch der Werte des Parameters der Perimeter-Regularisierung nu. Letzterer sollte problemabhängig nicht zu groß gewählt werden, um eine Optimalität des Zielgitters nicht stark zu verfälschen, wobei wir hierzu im nächsten Kapitel Untersuchungen anstellen. Hinzu kommen zudem die minimalen und maximalen Werte der Lamé-Parameter mu_min und mu_max, die nach 3.4 berechnet werden.
Weiterhin besitzt das Programm die Möglichkeit auszuwählen, ob ein L-BFGS-Verfahren oder ein Gradientenabstieg zur Optimierung verwendet werden soll. Dies lässt sich mit dem Parameter L_BFGS steuern, wobei L_BFGS = True das L-BFGS-Verfahren einschaltet. Wird dieses Verfahren verwendet, so muss man mit memory_length die Anzahl der gespeicherten und zur Berechnung verwendeten Gradienten und Deformationen einstellen. Weiterhin stellt curv_break, falls auf False eingestellt, die Möglichkeit zur Verfügung, bei Verletzung der Curvature Condition das Updaten auszulassen, jedoch einen Schritt mit den bisherigen Daten durchzuführen. Bei True wird bei selbiger Verletzung die Optimierung mit einer Fehlermeldung abgebrochen.
Zu den beiden möglichen Optimierungsverfahren lässt sich jeweils eine Linesearch durch Backtracking einschalten, indem man den Parameter Linesearch auf True setzt. Die weiteren Parameter und die Implementierung der Linesearch werden weiter unten bei 4.3.2 näher erklärt
Im Folgenden möchten wir den genauen Ablauf des Programms in Python

erläutern, wobei der schematische Ablauf in Abbildung **??** zu sehen ist.

Das Programm beginnt nach der Aktivierung über die Konsole mit der in 4.2.1 erklärten Initialisierung der Meshes. Je nach Einstellung findet die oben erklärte Pertubation des Startgitters statt. Weiterhin werden die lokal variierenden Lamé-Parameter, sowie die Zustandsfunktion im Zielgitter berechnet. Dies findet im Programm unter der Überschrift *Targetdata Calculation* statt. Es folgt der Einstieg in den Formoptimierungsalgorithmus, wobei unter *Interplation* die Daten auf das derzeitige Gitter zu einer FEniCS-Funktion interpoliert werden, für welche wir eine polynomielle Interpolation vom Grad 1 ausgewählt haben. Es werden anschließend die Zustands- und adjungierte Gleichung gelöst, sowie der Formgradient des derzeitigen Schrittes berechnet, was im Programm unter dem Schritt *State, Adjoint & Gradient Calculation* zu finden ist. Falls das Gradientenverfahren ausgewählt ist, wird zusätzlich als Deformation der negative Gradient initialisiert.
Nun findet im Programm die Unterscheidung zwischen dem L-BFGS- und dem Gradientenverfahren statt; der folgende Abschnitt ist lediglich für das L-BFGS-Verfahren relevant, im Gradientenverfahren springt der Algorithmus direkt zu der Linesearch 4.3.2.
Ein programmiertechnisch etwas aufwendiger Punkt ist die Berechnung der Curvature Condition. In unserem Setting ist diese errechnet durch

$$a(S_k, \mathcal{T}_{S_k}^{-1}(\nabla U_{k+1}) - \nabla U_k) = D\mathcal{J}(\Omega_{k+1})[\mathcal{T}_{S_k}(S_k)] - D\mathcal{J}(\Omega_k)[S_k] > 0. \quad (4.3)$$

Hierbei bedeutet $\mathcal{T}_{S_k}$ den Transport der Tangentialvektoren über die Deformation S_k, ihr Inverses den Rücktransport, wie in Algorithmus 3.17. Dies vermeidet die Lösung einer weiteren linearen Elastizitätsgleichung, wobei wir wie angesprochen die vereinfachten Transporte aus [26], Abschnitt 3, verwenden. Nichtsdestotrotz müssen wir zur Berechnung von $D\mathcal{J}(\Omega_{k+1})[\mathcal{T}_{S_k}(S_k)]$ eine Gitterverschiebung durchführen. Wir berechnen deshalb die Curvature Condition des k'ten Schrittes in Schritt Nummer $k+1$, da wir zuvor sowieso die passende Gitterverschiebung durchführen mussten. Die Deformation S_k aus dem Schritt k wird im Programm unter `last_defo` mitgeschleppt, der Wert der Ableitung $D\mathcal{J}(\Omega_k)[S_k]$ als Eintrag `curv_cond[1]` gespeichert. Wir wählen diese kompliziert wirkende Iteration, um zu häufiges verschieben der Meshes zu vermeiden, denn FEniCS verschiebt auch das Ausgangsgitter, wenn ein anderes Gitter unter selber Initialisierung verschoben wird. Dies könnte man vermeiden, indem man eine Deep Copy anlegt, was bei großen Gittern speicherintensiv sein kann. Hinzu kommen Rundungsfehler, deren Ausmaß wir jedoch nicht einschätzen können. Als Folge der um einen Schritt verschobenen Berechnung printen wir im L-BFGS-Verfahren die Werte der Schleife k in Schleife $k+1$, außerdem

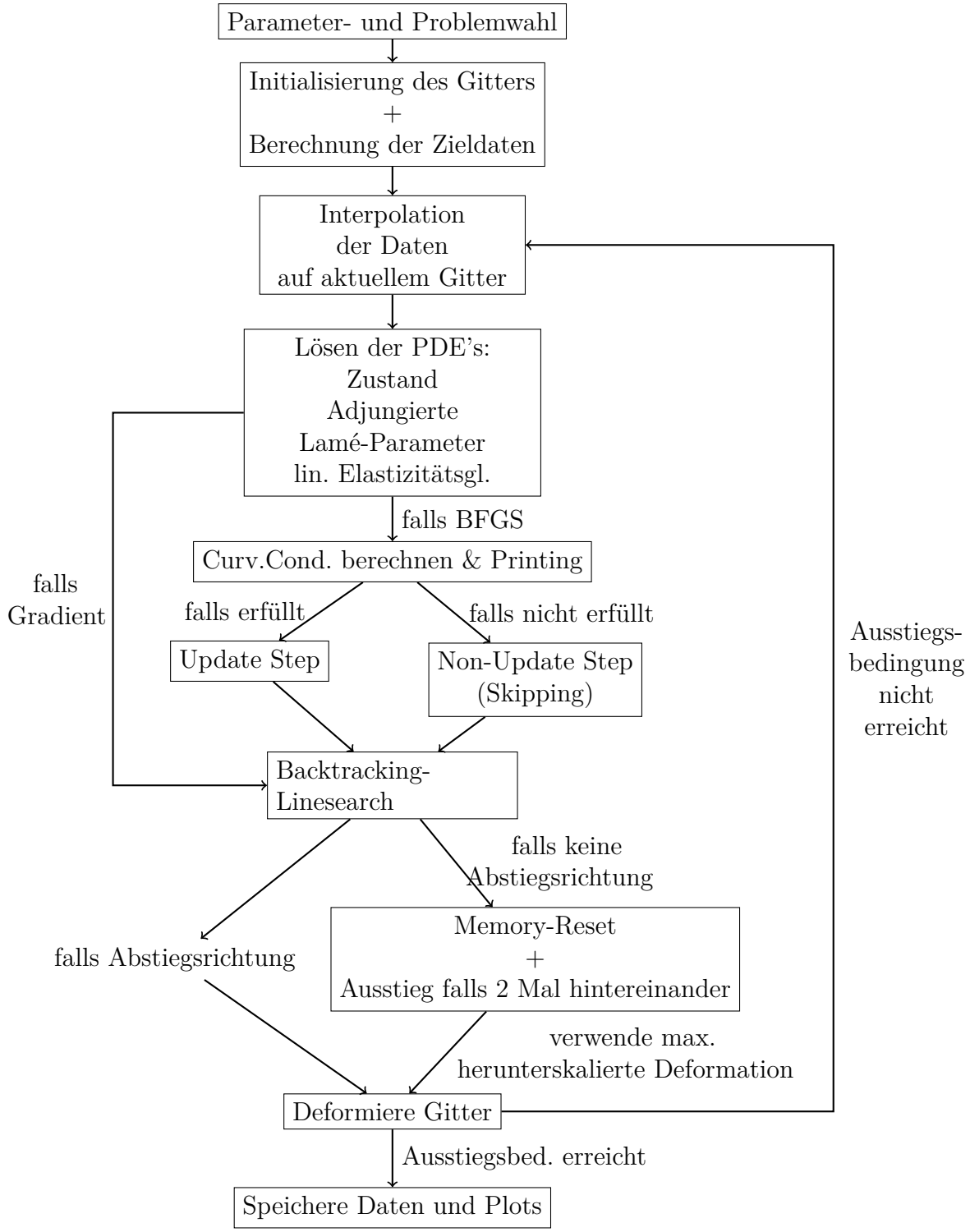


Abbildung 1: Schematische Darstellung des Hauptprogramms aus shape_main.py

wird die Memory aus selbigen Gründen erst dann geupdatet. Bei Einhaltung der Curvature Condition findet der Update unter dem Abschnitt *Update Step* statt, das Printen und die Berechnung der Curvature Condition unter dem Abschnitt *Curvature Condition & Printing*.
Im Falle, dass die Curvature Condition 4.3 nicht erfüllt ist, verwenden wir das Verfahren, die Memory nicht einem Update zu unterziehen, d.h. es wird lediglich ein Schritt mit schon vorhandenen Daten berechnet und angewandt. Dieses Skipping findet im Programm unter dem Abschnitt *Non Update Step* statt. Da der Update sowieso einen Schritt verzögert stattfindet, wird dieser einfach ausgelassen, und das Programm läuft weiter. Diese Strategie lässt sich mit der Einstellung des Parameter curv_break = True ausschalten, woraufhin der Algorithmus bei Verletzung der Curvature Condition aussteigt.

4.3.2 Die Backtracking-Linesearch

Nachdem die Gradienten und Deformationen berechnet wurden, erfolgt je nach Einstellung eine Linesearch mittels Backtracking. Diese wird gesteuert durch die Parameter Linesearch, shrinkage, c und start_scale. Ersterer schaltet die Linesearch bei Linesearch = True ein, shrinkage ist der Reskalierungsfaktor bei einem Backtracking-Schritt, c gibt den Faktor der Verbesserung im Zielfunktional im Vergleich zum derzeitigen Wert an, und start_scale gibt den Faktor zur Hochskalierung zum Beginn des Backtracking an.
Die Linesearch startet mit dem Berechnen der hochskalierten Deformation.

```
if(Linesearch):
  scale_parameter = start_scale

    zero_function = Function(VectorFunctionSpace(MeshData.mesh,
                                                 "P", 1, dim=2))
    current_value = bib.targetfunction(MeshData, zero_function,
                                       y_z, f_values, nu)
    S.vector()[:] = start_scale * S.vector()
    current_deriv = bib.shape_deriv(MeshData, p, y, z,
                                    f_values, nu, S)

    counterer = 0
```

Der Counter counterer zählt dabei die Anzahl der Reskalierungen der Deformation. Das eigentliche Backtracking findet in der gleich folgenden Schleife statt. Weiterhin ist in dieser Schleife die Strategie implementiert, bei maximalem Herunterskalieren des Deformationsfeldes die L-BFGS-Memory zu löschen und das Verfahren im jetzigen Punkt neu zu starten, wobei dies mit einer Mel-

dung zur Kenntnis gegeben wird. Falls das Verfahren im sofort darauf folgenden Schritt erneut maximal herunterskaliert, wird das Optimierungsverfahren mit einer Fehlermeldung abgebrochen. Hierzu wird der Counter Resetcounter verwendet.

```
while(bib.targetfunction(MeshData, S, y_z, f_values, nu)
      >= current_value):

  scale_parameter = shrinkage * scale_parameter
    S.vector()[:]   = shrinkage * S.vector()

    counterer = counterer + 1

  if(counterer >= 20):
      print("Had to break, restarting L-BFGS!")

        bfgs_memory.gradient     = np.zeros([memory_length, 2 *
                                     MeshData.mesh.num_vertices()])
        bfgs_memory.deformation = np.zeros([memory_length, 2 *
                                    MeshData.mesh.num_vertices()])
        bfgs_memory.step_nr      = 0
        Resetcounter             = Resetcounter + 1
        S.vector()[:]            = np.zeros(2*MeshData.mesh.
                                           num_vertices())
        break

        if(counterer < 20):
            Resetcounter = 0

        last_defo = S.vector().get_local()

if(Resetcounter >= 2):
    print("Reboot didn't help, quitting L-BFGS optimization!")
    break
```

Es ist außerdem möglich, statt der einfachen Backtrackingbedingung, die sogenannte *Armijo-Bedingung* mit Hilfe eines Backtracking-Verfahrens zu implementieren, siehe [28], Gleichung (3.4). Dies würde mit einer Schleife der Form

```
while(bib.targetfunction(MeshData, S, y_z, f_values, nu)
      > current_value + c*scale_parameter*current_deriv):
   #Schleifenaktionen von oben
```

funktionieren, wobei der Befehl targetfunction den Wert des Zielfunktionals im Gitter berechnet, welches durch Verschieben mit der Deformation S entsteht,

und dieses wieder zurück deformiert. Diese Schleife haben wir in auskommentierter Form im Code eingefügt, so dass der Benutzer diese durch entfernen der Auskommentierung verwenden kann.
An dieser Stelle sei erwähnt, dass wir die nach [9], Abschnitt 5, beschriebene Methode zur Einhaltung der Curvature Condition versucht haben zu implementieren. Dieses Verfahren wird *Powell-Relaxation* genannt, und ersetzt mit Hilfe eines Kriteriums, welches die BFGS-Approximierende B_k aus Definition 30 zur Berechnung verwendet, die Differenz der Gradienten y_k aus Definition 30. Das die Differenz der Gradienten ersetzende Vektorfeld würde dann die Curvature Condition erfüllen, und somit ein positiv definiten Update B_{k+1} garantieren. Zum einen haben wir dies nicht implementiert, da zur Berechnung B_k selbst benötigt würde, wir aber den 2-Schleifen-Algorithmus 3.17 verwenden, um lediglich die Anwendung der Inversen B_k^{-1} zu berechnen. Weiterhin führt dieses Verfahren zu einem sogenannten *Blow-Up* der Hesse-Approximation, weshalb das Verfahren dann neu gestartet werden müsste, und somit sich die Frage nach dem Nutzen im Vergleich zum Aufwand stellt. Aus diesen Gründen haben wir die Powell-Relaxation nicht implementiert.

4.3.3 Der Output

Abschließend kommen wir im Kapitel der Implementierung zum Output des Programms. Zu Beginn wird bei Aufruf des Programms ein Outputordner im Verzeichnis des Programms mit Namen `Output` erzeugt, falls dieser noch nicht vorhanden ist. Dies geschieht mit der implementierten Funktion `create_outputfolder()`. In diesen Ordner wird für jedes Aufrufen des Programms, sprich für jedes Optimierungsproblem, ein eigener Unterordner mit dem Namen erstellt, welcher nach dem exakten jetzigen Datum bis auf die Sekunde benannt wird. Beispielsweise würde ein Outputordner, welcher am 11.09.2001 um 8:46 und 0 Sekunden erzeugt wurde, den Namen

`20010911_084600`.

erhalten. Dadurch wird immer ein eindeutig bestimmter, neuer Outputordner erzeugt. In diesem werden die Gitter der einzelnen Iterationen mit den zugehörigen Formen, sowie die Daten des Zielgitters in `.pvd`-Dateien gespeichert, welche beispielsweise mit dem Programm *Paraview*[4] visualisiert werden können. Zusätzlich wird bei erfolgreichem Durchlaufen der Optimierung ein

[4]Wir verwenden für die Erzeugung der Bilder im nächsten Kapitel Paraview 5.0.1.

Konvergenzplot gespeichert, welcher die Abbildungen

$$k \mapsto \log(d_{shopt}(\Omega_k, \Omega_{target})$$
$$k \mapsto \log(||D\mathcal{J}(\Omega_k)||)$$

zeigt, wobei die Norm wie unter 4.1 erklärt berechnet wird. Um eine Analyse der Verfahren besser durchführen zu können, werden weiterhin relevante Größen in einer .cvd-Datei gespeichert. Diese Daten werden in der selben Anordnung gespeichert, wie sie im Laufe des Programms auch für den Benutzer in der Konsole geprintet werden. Schematisch sieht dies wie folgt aus:

Iteration ||f_elas||_L2 J = j + j_reg ||U||_L2 Curv. Cond. Meshdistance

Falls das Gradientenverfahren ausgewählt wurde, so wird die Spalte mit der Curvature Condition weggelassen. Diese Datei lässt sich im Rahmen des Postprocessings bequem mit Programmen wie *Excel* oder *RStudio* bearbeiten. Sollte das Verfahren divergieren, etwa falls die Curvature Condition einige Schritte nicht erfüllt ist, so findet keine Speicherung der Konvergenzgraphen statt. Die .pvd-Dateien werden trotzdem gespeichert, die .csv-Datei ist leer, wobei man dies durch Erstellen und Speicherung einer neuen .csv-Datei in jedem Schritt umgehen könnte. Im nun folgenden und letzten Kapitel werden wir solche Analysen für einige von uns ausgewählte Routinen vorstellen.

5 Resultate

Wie angekündigt möchten wir in diesem abschließenden Abschnitt die Resultate aus unserer Programmierung analysieren. Wir haben mehrere Problemgitter und Kombinationen aus Verfahren getestet, und dabei die Daten aufbereitet. Diese Daten werden samt dem Programm bei der Abgabe zur Verfügung gestellt. Wir haben uns für die Analyse der Verfahren für zwei Testprobleme entschieden. Zum einen soll ein kleiner Kreis zu einem großen Kreis deformiert werden, wobei die Kreismittelpunkte jeweils gleich bleiben. Somit handelt es sich bei diesem Problem um eine Deformation einer konvexen Form ohne große Translation. Zum anderen soll ein kleiner Kreis zu einer nierenartigen Form deformiert werden. In diesem Problem findet eine verhältnismäßig starke Translation statt, außerdem ist die Zielform ein nicht konvex, was dieses Problem deutlich schwieriger als das erste macht. In beiden Fällen sind die Formen in dem Einheitsquadrat im $\mathbb{R}^2$ eingebettet. Diese sind zu sehen in Abbildung 2.

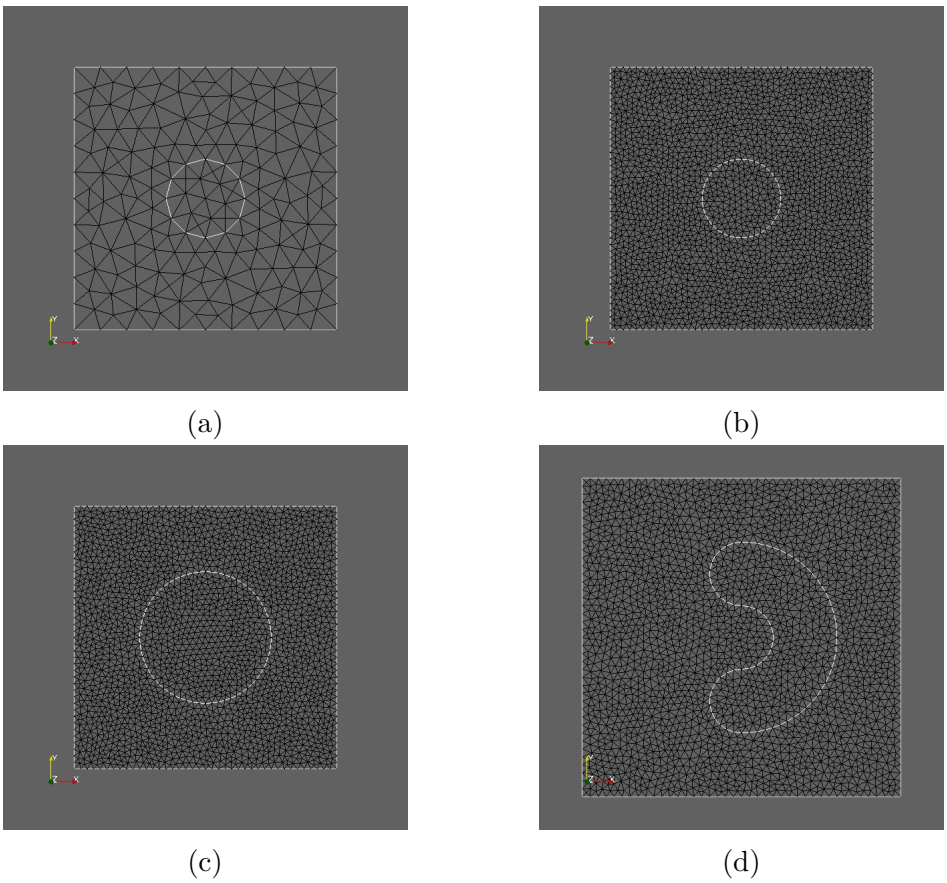
Die Parameter, welche bei der Analyse in Betracht kommen, sind als Standard auf folgende Werte gesetzt:

Gitterfeinheit:	0.1 (normal) , 0.025 (fein)
Lamé-Parameter:	0.0 (min) , 30.0 (max)
Perimeter-Reg.:	0.00001
Funktionswerte für Zustand:	-10.0 (außen) , 100.0 (innen)
Memory-Length:	60
Toleranz für Ausstieg:	0.0008
Backtracking:	5.0 (start_scale), 0.5 (shrinkage), 0.999 (c)

Wir werden sowohl das L-BFGS-Verfahren, als auch das Gradientenverfahren vergleichen. Außerdem wird jeweils die Linesearch an- und ausgeschaltet, sowie alle oben aufgeführten Parameter, bis auf die Funktionswerte die Zustandsgleichung im Zielgitter, den Verbesserungsfaktor c, und den Skalierungsfaktor shrinkage der Backtracking-Linesearch, variiert. Die Verwendung der Memory-Length von 60 bewirkt, dass es sich in allen folgenden Fällen der Verwendung des L-BFGS-Verfahrens eigentlich um ein BFGS-Verfahren handelt, da die gesamte Historie zur Berechnung der Hesse-Approximation benutzt wird.

5.1 Problem: kleiner zu großem Kreis

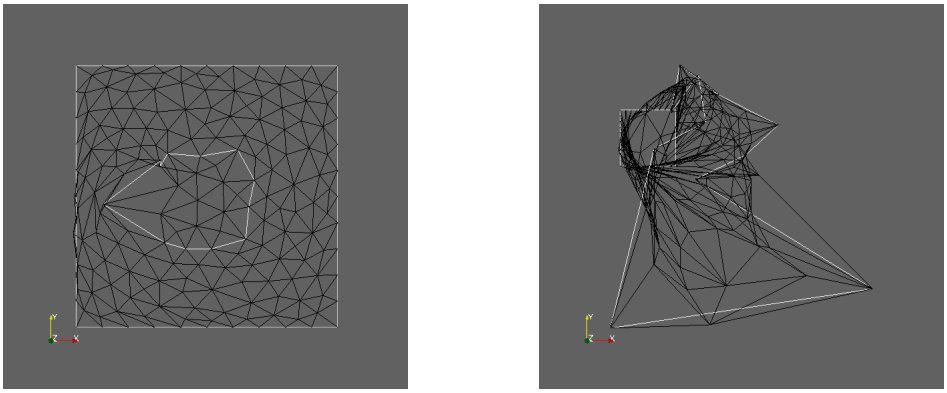
Für das Problem der Deformation zum großen Kreis betrachten wir vor dem BFGS-Verfahren zunächst das Gradientenverfahren. Alle Zeiten und Anzahlen der benötigten Schritte der Verfahren finden sind in Tabelle 1. Wir erhalten bei dem groben Gitter bei Verwendung des Gradientenverfahrens ohne Linesearch Konvergenz nach fast 800 Schritten. Schaltet man das Backtracking ein, so erhält man ebenfalls Konvergenz, jedoch bei gut halber Anzahl der Schritte. Dies legt nahe, das die Schrittweite durch das anfängliche Hochskalieren der Deformation optimaler wird, der Gradient also der Norm nach relativ klein war. Bei Verfeinerung des Gitters um das 4-fache erhalten wir ebenfalls ohne Verwendung der Linesearch beim Gradientenverfahren Konvergenz nach ca. 650 Schritten, wobei wieder durch Verwendung des Backtracking eine Beschleunigung zur Konvergenz stattfindet. Die Konvergenzplots finden sich in Abbildung 6. Erhöht man das anfängliche Hochskalieren vom Faktor 5 auf 20, so erhält man im feinen Gitter Konvergenz schon nach 120 Schritten. Die Geschwindigkeit zur Konvergenz hat sich also im Vergleich zum Gradientenverfahren ohne Backtracking also um das 5-fache gesteigert, wobei sich die erzeugten Gitter



(a) (b)

(c) (d)

Abbildung 2: (a) Ausgangsform kleiner Kreis auf grobem Gitter (b) Ausgangsform kleiner Kreis auf feinem Gitter (c) Zielform großer Kreis auf feinem Gitter (d) Zielform Broken Donut auf feinem Gitter



(a) (b)

Abbildung 3: (a) Gitter nach mehrmaligen Schritten unter Verletzung der Curvature Condition und Non-Update Schritten (b) der darauf folgende Schritt, Zerstörung des Gitters

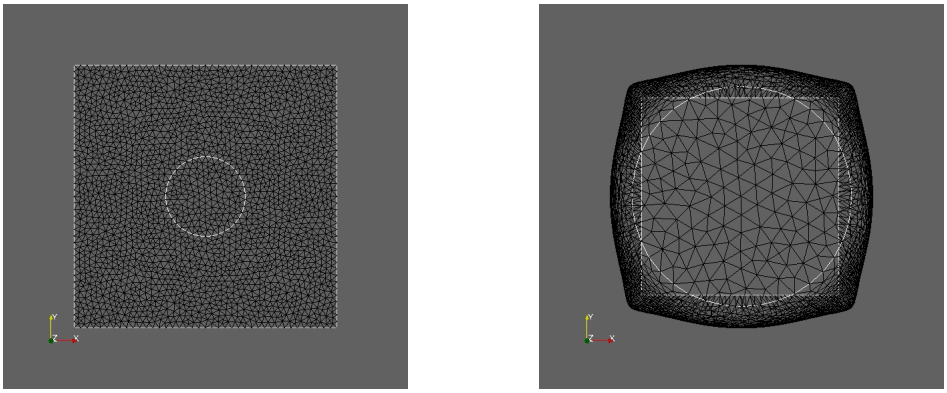
bei Ausstieg des Verfahrens mit oder ohne Linesearch und Veränderung des Hochskalierungsfaktors so gut wie nicht unterscheiden.

Problem	Zeit Gradientenv.	Zeit BFGS-Verf.	Schritte Gradientenv.	Schritte BFGS-Verf.
großer Kreis (grob)	13,7s	4,8s	33	6
großer Kreis (fein)	1min 31,3s	17,2s	52	7
Broken Donut (grob)	43,3s*	6,6s	109*	9
Broken Donut (fein)	27min 4,6s	1min 25,4s	910	27

* *Gradientenverfahren steigt bei Genauigkeit 0.0033 aus, weil Linesearch keine Abstiegsrichtung findet*

Tabelle 1: Rechendauern und Schritte der Verfahren (mit Backtracking-Linesearch)

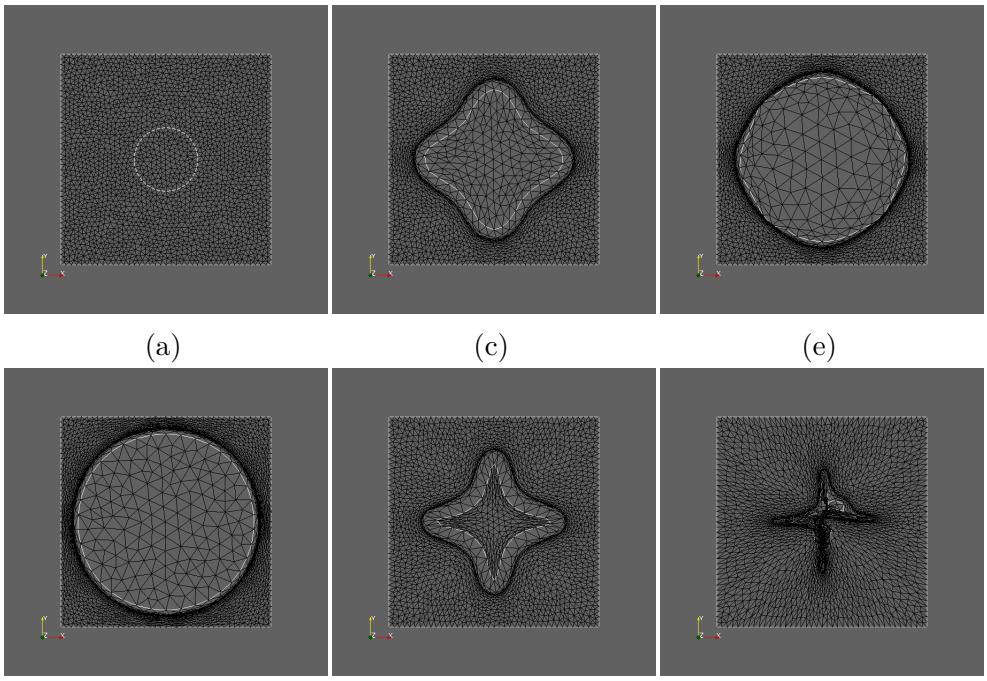
Findet das BFGS-Verfahren ohne Linesearch Anwendung, so erhält man auf dem groben Gitter keine Konvergenz. Das Verfahren updatet die BFGS-Memory nach mehreren Verletzungen der Curvature Condition nicht, und deformiert das Gebiet schließlich bei Schritt 26 bis zur Unbrauchbarkeit, was in Abbildung 3 zu sehen ist. Man beachte die starke Entartung der kaum sichtbaren Zellen kurz vor der Zerstörung des Gitters, welche man im Zoom gut erkennt. Auch bei Verfeinerung des Gitters um das 4-fache erhält man keine Konver-



(a) (b)

Abbildung 4: (a) 1. Schritt des BFGS-Verfahrens: Gradientenschritt bei feinem Gitter (b) 2. Schritt: BFGS-Schritt mit Entartung des feinen Gitters

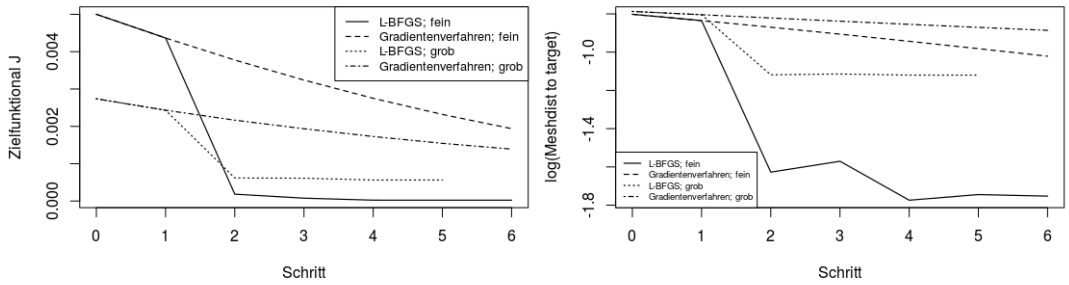
genz des BFGS-Verfahrens, sondern erreicht die Zerstörung des Gitters nach Verletzung der Curvature Condition schon im zweiten Schritt, bei der die Form das Einheitsquadrat verlässt, siehe Abbildung 4. Man beachte, das dieses mal die Richtung des Schrittes in Richtung Optimum geht, jedoch viel zu lang ist, anders als in Abbildung 3, wo zuvor mehrmalige falsche Schritte bei Verletzung der Curvature Condition das grobe Gitter zerstören. Diese Beobachtung macht erneut deutlich, wie wichtig eine effektive Schrittweitensteuerung bei der Implementierung der Verfahren ist.
Wir haben auch versucht, die Gitterzerstörung beim BFGS-Verfahren durch Modifikation der lokal variierenden Lamé-Parameter zu begegnen. Alle bisher gezeigten Gitter hatten die Wahl des minimalen Lamé-Parameters von 0. Um zu beobachten, ob die Verfahren instabiler werden, wenn man konstante Lamé-Parameter wählt, haben wir diese konstant auf 30 gesetzt. Man erhält den Output 5, wobei bei Schritt 5 und 6 die Curvature Condition verletzt sind, und keine Linesearch verwendet wurde.
Ein komplett anderes Verhalten stellt sich bei Verwendung der Backtracking-Linesearch ein; in diesem Fall konvergiert das L-BFGS-Verfahren für beide Gitterfeinheiten. Zudem fällt eine erhebliche Steigerung der Konvergenzgeschwindigkeit im Vergleich zum Gradientenverfahren auf; schon nach 5 Schritten auf dem groben, und nach 6 auf dem feinen Gitter. Die entsprechenden Konvergenzplots für die Verfahren mit Linesearch auf grobem und feinem Gitter sind zu sehen in 6.



(a) (c) (e)

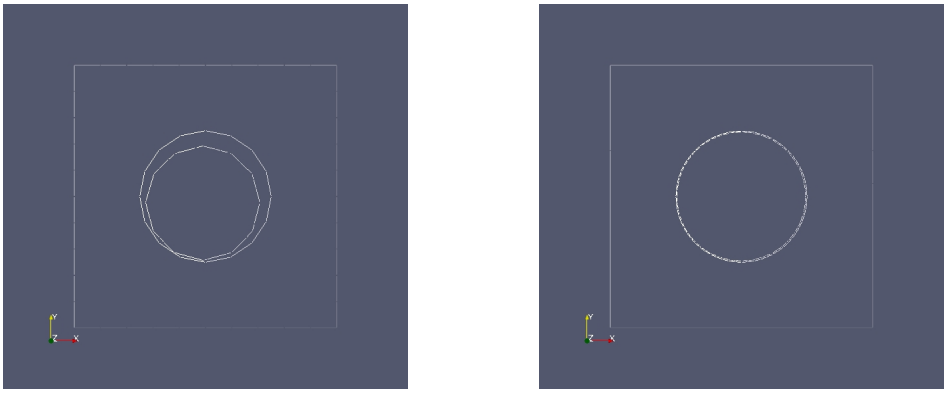
(b) (d) (f)

Abbildung 5: BFGS-Verfahren ohne Schrittweitensteuerung bei konstanten Lamé-Parametern mit Wert 30; die ersten 6 Schritte



(a) (b)

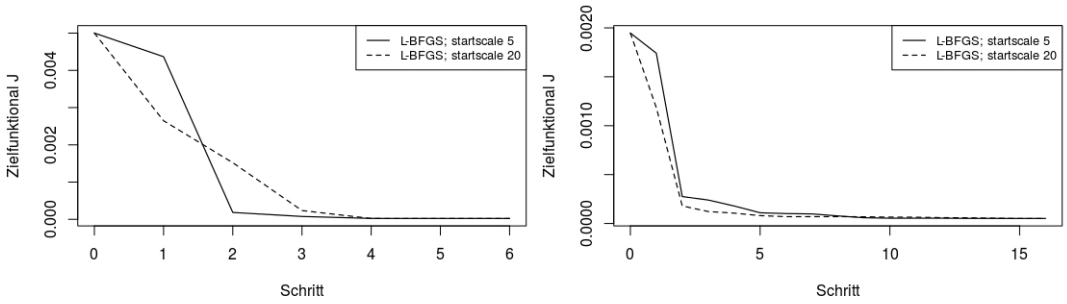
Abbildung 6: kleiner Kreis auf großer Kreis (a) Werte des Zielfunktionals bei den Verfahren mit Linesearch; unterschiedliche Gitterfeinheiten (b) logarithmierte Meshdistanz bei Verfahren mit Linesearch; unterschiedliche Gitterfeinheiten



(a) (b)

Abbildung 7: (a) L-BFGS mit Backtracking bei grobem Gitter. Abweichung bei Ausstieg sichtbar. (b) L-BFGS mit Backtracking bei feinem Gitter. Abweichung bei Ausstieg nicht erkennbar.

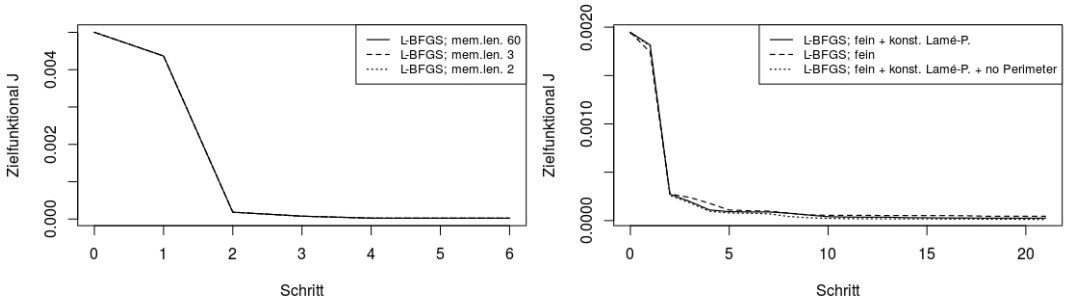
Die bei dem groben Gitter erzeugte optimale Form besitzt noch einen mit bloßem Auge sichtbaren Abstand zur Zielform. In diesem Fall hat unser L-BFGS-Algorithmus bei dem Backtracking maximal herunterskaliert, d.h. keine Abstiegsrichtung gefunden und ist ausgestiegen. Damit scheint ein lokales Minimum erreicht, zu sehen in Abbildung 7 (a). Diese Vermutung liegt nahe, da die zugehörigen Werte der Norm der Formableitung nrm_f_elas gegen Null streben, hierzu verweisen wir auf die mitgelieferten Daten. Bei Verwendung des feinen Gitters ist bei selbem Ausstiegskriterium kein Abstand zur Zielform sichtbar, was man in Abbildung 7 (b) sehen kann.
Wir haben durch Veränderung der Perimeter-Regularisierung untersucht, ob sich durch diese der Abstand zur Zielform erklären lässt. Die Vermutung ist, dass die optimale Form etwas kleiner als die Zielform ist, da die Perimeter-Regularisierung kleinere Formen favorisiert. Die in diesem Fall entstehende optimale Lösung ist in diesem Fall symmetrisch mit gleichem Mittelpunkt wie die Zielform, jedoch besitzt sie fast den selben Abstand wie die optimale Form mit der Perimeter-Regularisierung aus Abbildung 7 (a). Für die exakten Bilder verweisen wir auf die mitgelieferten .pvd-Dateien.
Kontrollieren wir im Falle des BFGS-Algorithmus auf Veränderung bei Erhöhung des Hochskalierungsfaktors starscale des Backtracking, so stellen wir fest, dass die ersten Schritte stärkere Verbesserungen bringen. Sobald sich jedoch eine gewisse Memorygröße, und damit eine Hesse-Approximation, aufgebaut haben, verschwinden diese Effekte wieder, siehe Abbildung 8. Dieses Verhal-



(a) (b)

Abbildung 8: Zielfunktional bei Variation des Skalierungsparameters start_scale des Backtracking für das L-BFGS-Verfahren auf feinen Gittern der Probleme (a) kleiner Kreis auf großen Kreis (b) kleiner Kreis auf "Broken Donut"

ten sollte auftreten, da BFGS-Schritte mit aufgebauter Hesse-Approximation skalierungsinvariant sind.
Zudem haben wir einen Vergleich zwischen der vollen BFGS-Methode, welche wir durch die memory_length von 60 erzeugen, und der echten Limited-Memory-BFGS-Methode, mit memory_length von 2 und 3, im Falle des feinen Gitters gezogen. Interessanterweise stellt sich hier überhaupt kein Unterschied hinsichtlich Konvergenzgeschwindigkeit und erzeugte Werte im Zielfunktional ein, was man in Abbildung 9 (a) erkennen kann. Ähnliche Beobachtungen machten die Autoren von [26] bei elliptischen und parabolischen Problemen, siehe das genannte Paper, Abschnitt 3. Dies lässt sich nach unserer Vermutung dadurch erklären, dass für einen korrekten BFGS-Schritt ein Term fehlt. Würde dieser ergänzt, so hoffen wir auf eine Verbesserung des Algorithmus bei höherer memory_length auftreten sollte. Außerdem lässt sich in den Diagrammen keine asymptotische superlineare Konvergenz nachweisen, welche sich bei korrektem Update möglicherweise einstellt.
Neben der bloßen Bedingung einer Verbesserung im Zielfunktional bei dem Backtracking besteht die Möglichkeit, die Armijo-Bedingung für die Linesearch zu verwenden. Diese haben wir nach [28], wie im Unterabschnitt 4.3.2 gezeigt, implementiert. Verwendet man diese, so konvergiert die L-BFGS-Methode nicht mehr, in einigen Fällen konvergiert das Gradientenverfahren. Jedoch würde wir eher behaupten, dass dies *trotz* der Armijo-Bedingung konvergiert. Da diese



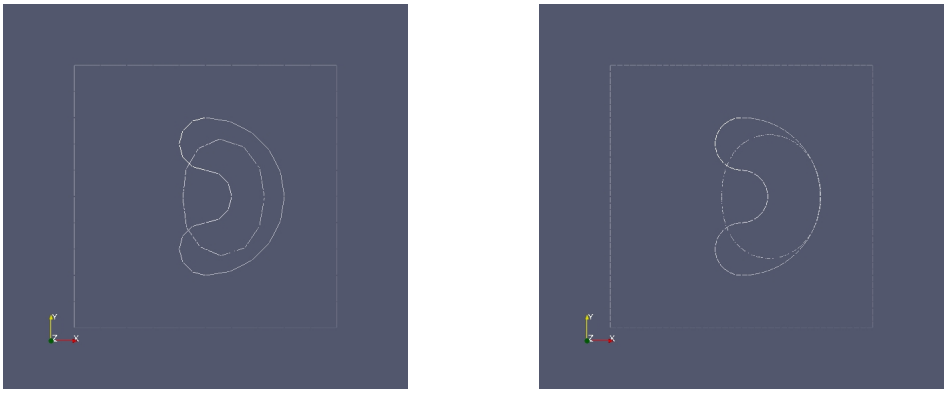
(a) (b)

Abbildung 9: (a) Variation des L-BFGS-Parameters memory_length bei feinem Gitter des Problems kleiner Kreis auf großer Kreis. Zu sehen sind 3 verschiedene Graphen, numerische Abweichungen treten erst ab Schritt 5 ab der 3 Nachkommastelle auf, weshalb diese fast gleich aussehen.
(b) Änderung der Lamé-Parameter und Entfernen der Perimeter-Regularisierung auf feinem Gitter des Problems kleiner Kreis auf "Broken Donut".

Bedingung die Verbesserung mit Hilfe eines linearen Modells in der aktuellen Form betrachtet, wäre ein aus der Formoptimierung legitimes anderes Modell vorstellbar, um verbesserte Bedingungen zur Linesearch zu erzeugen. Hierzu hat uns leider die Zeit gefehlt.
Bei der Berechnung des Formgradienten, wie im Unterabschnitt 4.2.2 erklärt, haben wir ein Verfahren verwendet, welches in der linearen Elastizitätsgleichung die Punkte, welche nicht Träger des inneren Randes sind, auf Null setzt. Schaltet man diese Nullsetzung aus, was durch die Wahl des Parameters zeroed = False möglich ist, so zerstört dies die Konvergenz des Verfahrens. Sowohl das L-BFGS, als auch das Gradientenverfahren divergieren in allen von uns getesteten Beispielen. Somit tritt der selbe Effekt wie in [27], welcher dort in Abschnitt 5, Fig. 2 zu sehen ist. Würde dieser Effekt durch ein Rundungsrauschen hervorgerufen sein, so vermuten wir, dass dieses weniger drastisch sichtbar würde. Bei uns ist schon nach den ersten Schritten das Gitter bis zur Unbrauchbarkeit entartet. Eine befriedigende Erklärung hierfür haben wir leider nicht.

5.2 Problem: kleiner Kreis zu 'Broken Donut'

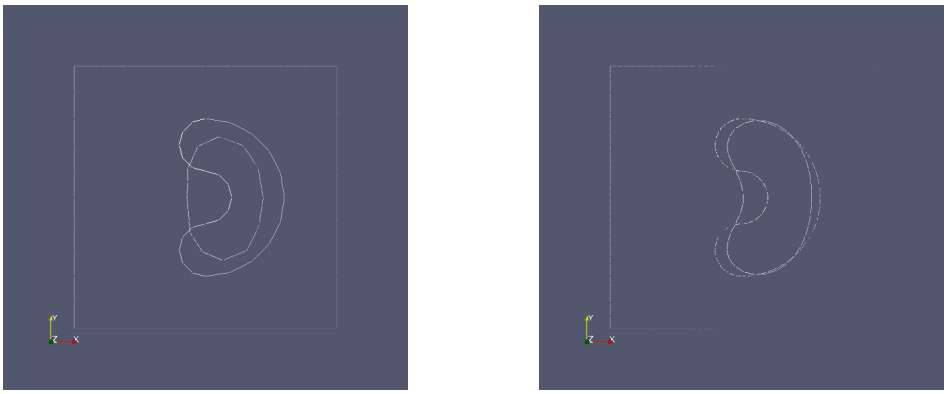
Wir haben die oben genannten Beobachtungen auch für das schwierigere Problem der nierenförmigen, nicht konvexen Form gesammelt. Die Ausgangsform, welcher den kleinen Kreis in Abbildung 2 (a) und (b) darstellt, bleibt gleich, wir wechseln also nur die Zielform, die auch in Abbildung 2 (d) zu sehen ist.
Anders als bei dem einfacheren ersten Problem, erhalten wir für ein Gradientenverfahren ohne Backtracking-Linesearch, in den Fällen sowohl des groben, als auch des feinen Gitters, nach ca. 1200, respektive ca. 600 Schritten zwar Konvergenz, jedoch nicht zum globalen Optimum des Zielgitters. Wir vermuten, dass es sich hierbei um lokale Minima handelt, weil die Norm der Formableitung nrm_f_elas gegen 0 geht, wobei wir auf die Daten verweisen. Die resultierende optimale Form bleibt konvex, trotz des nicht konvexen Zielgitters, siehe Abbildung 10.
Durch das Verwenden der Backtracking-Linesearch schafft es der Algorithmus näher an das globale Optimum zu gelangen. Dies gelingt jedoch ausschließlich bei dem feinen Gitter. Im Falle des groben Gitters erreichen wir auch mit Linesearch die selbe nicht konvexe Form, siehe Abbildung 11, wobei eine leichte Verbesserung zu sehen ist. Dies zeigt, dass das Verfahren abhängig von der Gitterfeinheit ist, wobei Verfeinerungen Verbesserungen liefern. Für die Zeiten und Schritte verweisen wir auf Tabelle 1.



(a) (b)

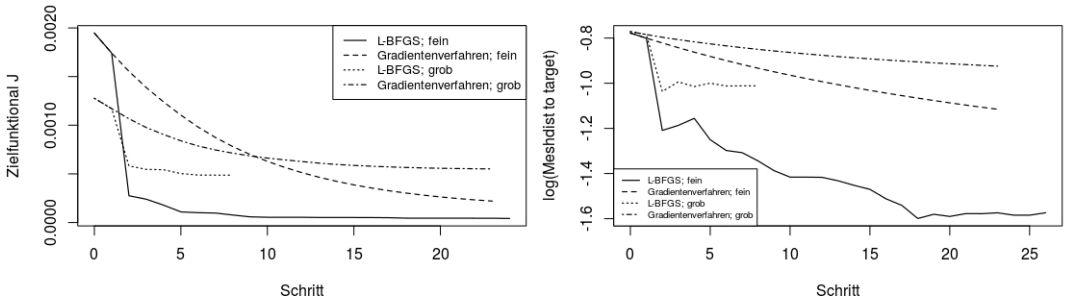
Abbildung 10: (a) Optimale Form und Zielform bei Gradientenverfahren auf grobem Gitter (b) Selbiges auf feinem Gitter

Außerdem beobachten wir für das Gradientenverfahren, wie im ersten Problem, ein Skalieren der Konvergenzgeschwindigkeit durch das Hochskalieren der Suchrichtung bei der Backtracking-Linesearch. Wir sparen uns an dieser Stelle die Angabe eines Plots, da dieses Phänomen bekannt ist. Zudem haben wir in diesem Problem die gleichen Feststellungen bei Verwendung der Armijo-Bedingung bei der Linesearch gemacht, wie bei dem ersten Problem zuvor.
Bei Verwendung des L-BFGS-Verfahrens ohne Linesearch bemerken wir, ähnlich wie zu dem ersten Beispiel, dass sowohl beim groben, als auch beim feinen Gitter eine zu Abbildung 4 ähnliche Divergenz stattfindet, weshalb wir uns Graphiken hierzu sparen. Das Gitter ist nach nur wenigen Schritten, nachdem die Curvature Condition verletzt wurde, bis zur Unkenntlichkeit verformt, wobei das Verfahren auch nicht die lokalen Minima 11 oder 10 erreicht.
Interessanterweise verbessert sich das Verhalten des L-BFGS-Verfahrens ohne Linesearch, wenn man statt lokal variierenden Lamé-Parametern konstante Lamé-Parameter mit Wert 30 wählt. Zuvor divergierte das Verfahren bei dem feinen Gitter nach nur 5 Schritten unter Verletzung der Curvature Condition der letzten 3 Schritte. Bei konstanten Lamé-Parametern findet zwar auch Divergenz statt, jedoch erst nach 14 Schritten, wobei zuvor sogar ein optimales Ergebnis, welches dem Gradientenverfahren mit Linesearch auf dem feinen Gitter nahe kommt, erreicht wird. Zu sehen sind ausgewählte Schritte in Abbildung 13. Man beachte jedoch den Vergleich zu dem Gitter, welches bei dem L-BFGS-Verfahren mit Linesearch und variierenden Lamé-Parametern



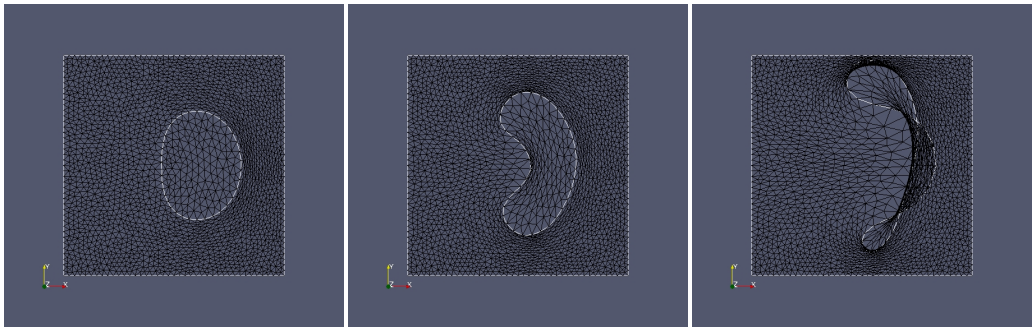
(a) (b)

Abbildung 11: (a) Optimale Form und Zielform bei Gradientenverfahren mit Linesearch auf grobem Gitter (b) Selbiges auf feinem Gitter



(a) (b)

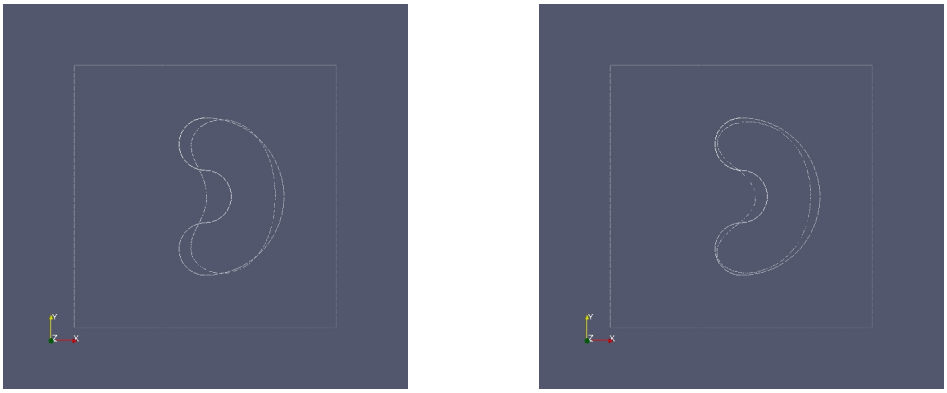
Abbildung 12: kleiner Kreis auf "Broken Donut"(a) Werte des Zielfunktionals bei den Verfahren mit Linesearch; unterschiedliche Gitterfeinheiten (b) logarithmierte Meshdistanz bei Verfahren mit Linesearch; unterschiedliche Gitterfeinheiten



(a) (b) (c)

Abbildung 13: BFGS-Verfahren bei konstanten Lamé-Parameter von 30 (a) Schritt 6 (b) gute Form bei Schritt 12; man beachte die starke Stauchung/ Streckung der Zellen (c) Zerstörung des Gitters bei Schritt 14

erzeugt wird, bei dem geringere Entartung der Zellen des Gitters zu beobachten sind. Wir würden an dieser Stelle vorschlagen, ein Kriterium für die Entartung des Gitters, etwa dem Quotienten aus maximalem und minimalem Zellvolumen, zu verwenden, um bei Überschreitung eines festgelegten Entartungsgrades ein neues Gitter mit der aktuellen Form zu initialisieren. Dies ließe sich auch mit lokal variierenden Lamé-Parametern, sowie mit adaptiven Gittermethoden, kombinieren.
Verwendet man zusammen mit dem L-BFGS-Verfahren die Backtracking-Linesearch, so erhalten wir ähnliche Ergebnisse wie in dem ersten Problem, nämlich eine deutliche Erhöhung der Konvergenzgeschwindigkeit im Vergleich zum Gradientenverfahren auf dem feinen Gitter. Auch auf dem groben Gitter ist eine erhebliche Erhöhung der Konvergenzgeschwindigkeit zu beobachten, wobei auch hier lediglich das lokale Minimum des Gradientenverfahrens erreicht wird. Auf dem feinen Gitter wird, ähnlich dem Gradientenverfahren mit Linesearch, eine sehr gute Lösung erreicht, nahe des globalen Optimums. Zu sehen sind die Ergebnisse des BFGS-Verfahrens mit Linesearch in Abbildung 14, für die Zeiten und Schritte verweisen wir erneut auf Tabelle 1. Bei dem Gradientenverfahren ist bei beiden untersuchten Problemen zu erkennen, dass eine deutliche Verfeinerung des Gitters keine Verlangsamung der Konvergenz mit sich bringt, was ein Indiz für gitterunabhängige Konvergenz ist, siehe Abbildung 12 und 6. Das BFGS-Verfahren weist zu schnelle Konvergenz auf, so dass wir hier keine Aussage über die gitterunabhängige Konvergenz machen können.
Die Curvature Condition wird ab Schritt 17 verletzt, wobei schon in diesem



(a) (b)

Abbildung 14: L-BFGS-Verfahren mit Linesearch (a) Form bei Ausstieg auf grobem Gitter (b) Form bei Ausstieg auf feinem Gitter

Schritt eine fast vom Ergebnis kaum zu unterscheidbare Lösung erzeugt wird. Die Ausstiegsbedingung war somit lediglich zu stark gewählt, was man auch deutlich an dem Graphen aus Abbildung 12 des feinen Gitters sehen kann.
Weiterhin stellen wir fest, dass durch das Weglassen der Perimeter-Regularisierung die Konvergenz in diesem Problem erhalten bleibt, siehe den Graphen 9 (b). Lässt man zusätzlich die Lamé-Paramter nicht lokal variieren, so bleibt auch hier die Konvergenz erhalten, das Verfahren benötigt jedoch etwas mehr als die doppelte Anzahl an Schritten bis die Toleranz an die Formableitung unterschritten wird. Betrachtet man die Konvergenzgraphen, fällt jedoch auf, dass dies an dem Ausstiegskriterium liegt, welches etwas zu stark gewählt ist. Somit funktioniert das Verfahren auch unter Anwendung konstanter Lamé-Parameter, sowie bei komplettem Weglassen der Perimeter-Regularisierung, erneut zu sehen in Abbildung 9 (b). Allerdings liegt in allen Fällen ohne Linesearch Divergenz vor. Werden lediglich die Lamé-Parameter konstant gehalten, so tritt keine Änderung des Konvergenzverhaltens auf, was auch in Abbildung 9 (b) zu sehen ist. Allerdings weisen die Zellen ohne lokal-variierende Lamé-Parameter einen höheren Entartungsgrad auf.
Außerdem beobachten wir, dass bei einer Memory-Length von 3, sprich bei einem echten Limited-Memory Ansatz, die Anzahl der benötigten Schritte zur Konvergenz von 26 auf 35 auf dem feinen Gitter steigt, jedoch bei dem groben Gitter gleich bei 8 Schritten bleibt.
Zuletzt möchten wir anmerken, dass bei Betrachtung aller Konvergenzgraphen des BFGS-Verfahrens, sowohl des ersten Problems 6, als auch des zweiten Pro-

blems 12, eine erhebliche, sprungartige Verbesserung im zweiten Schritt erfolgt. Wir vermuten, dass dies durch die Verwandtschaft des L-BFGS-Verfahrens mit dem CG-Verfahren erklärbar ist. Die Autoren von [19] zeigen für die Minimierung quadratischer Zielfunktionen im $\mathbb{R}^n$ unter gewissen Bedingungen, dass alle Mitglieder der Broyden-Klasse, somit auch das BFGS-Verfahren, bei vollen Updates nach nur n Schritten die Lösung erreichen. Verwenden die Autoren die Limited-Memory-Varianten, so verlieren alle Mitglieder der Broyden-Klasse diese Eigenschaft, mit einziger Ausnahme des L-BFGS-Verfahrens. Weiterhin zeigen die Autoren, dass bei der auch von uns angewandten Skipping-Strategie, dem Aussetzen von Updates bei Verletzung der Curvature-Condition, bei q-maligem Aussetzen immer noch ein Erreichen der Lösung nach nur $n+q$ Schritten vorliegt. Zwar haben die Autoren dies wie erwähnt für quadratische Funktionen im $\mathbb{R}^n$ gezeigt, unsere Beobachtungen bei den genannten Konvergenzgraphen legen jedoch ein ähnliches Verhalten in Kontext der Formoptimierung nahe.
Wir beenden nun die Analyse der von uns erzeugten Daten, und laden die Leser dieser Arbeit gerne dazu ein, dass beiliegende Programm bei Interesse selber einigen Tests zu unterziehen. Abschließend möchten wir einen kleinen Ausblick auf mögliche weiterführende Themen und einige Probleme geben.

5.3 Ausblick und Schlussbetrachtung

Wir haben mit der in Kapitel 4 vorgestellten Implementierung, und den Ergebnissen aus den Abschnitten 5.1 und 5.2, ein durchaus effektives Quasi-Newton-Verfahren in der Formoptimierung demonstriert. Die Konvergenzgeschwindigkeit im Vergleich zum Gradientenverfahren ist deutlich erhöht. Wir haben gesehen, dass eine Art der Schrittweitensteuerung, etwa dem bei uns implementierten Backtracking-Linesearch, für die Konvergenz des L-BFGS-Verfahrens nötig ist. An dieser Stelle gibt es Bedarf für weitere Entwicklungen; so wäre entweder eine im vorigen Kapitel angesprochene Verbesserung des Kriteriums für hinreichend gute Abstiege, die ein aus dem Formkalkül legitimes Modell für das Zielfunktional verwenden, denkbar. Diese könnte im Falle eines linearen Modells das Armijo-Kriterium auf Formräume verallgemeinern. Eine andere Variante wäre die Verwendung einer Art Trust-Region-Methode, mit dessen Hilfe auch eine globale Konvergenz erreicht werden könnte. Zudem kann sich mit diesem Ansatz die Möglichkeit bieten, durch das Vermeiden der Indefinitheit der Quasi-Newton-Approximation andere Verfahren, etwa den Broyden-SR1-Update, anwenden zu können. In den Konvergenzgraphen sind

außerdem deutliche Verbesserungen nach nur wenigen Schritten zu erkennen, welche wir zuvor versucht haben zu erklären, und eine Ähnlichkeit zum CG-Verfahren ausweisen. Eine Limited-Memory-Variante des CG-Verfahrens für die Formoptimierung wäre eine weitere Möglichkeit dem zuvor genannten Problemen zu begegnen. Außerdem haben wir keine asymptotisch superlineare Konvergenz beobachten oder zuverlässig widerlegen können. Um dies anzugehen, wäre eine Anpassung der BFGS-Updates wichtig. Wie sich diese konkret im Formraum realisieren lässt, ist jedoch bis dato unklar. Wir haben weiterhin Indizien für gitterunabhängige Konvergenz bei Untersuchung des Gradientenverfahrens bemerkt, so dass es sich lohnenswert erscheint, das Verhalten der Verfahren bei anderen Problemen und anderer Wahl der Bilinearform a und der damit erhaltenen Metrik g^S zu analysieren. Eine Möglichkeit hierfür würde die biharmonische Gleichung bieten. Abschließend bleibt noch das etwas ominöse Problem bei Erzeugung der Formgradienten, bei dem ohne Nullsetzen der Punkte ohne Träger am inneren Rand zur Lösung der linearen Elastizitätsgleichung die verheerenden Effekte auftreten, welche das Gitter zerstören und wir nicht befriedigend erklären können. Dies sind viele mögliche Ansätze für kommende Fortschritte im Bereich der Formoptimierung. Wir hoffen, mit dieser Arbeit einen kleinen Beitrag in die richtige Richtung gegeben zu haben, und bedanken uns bei Prof. Volker Schulz für die Möglichkeit diese Masterarbeit anzufertigen.

Literatur

[1] G. Geymonat. Trace Theorems for Sobolev Spaces on Lipschitz Domains. Necessary Conditions. 2007.

[2] H. Diab, R. Younes, P. Lafon. Survey of research on the optimal design of sea harbours. *International Journal of Naval Architecture and Ocean Engineering, Vol. 9, Issue 4, pp. 460-472*, 2017.

[3] Jan Sokolowski, Jean-Paul Zolesio. *Introduction to Shape Optimization*, volume 16. Springer-Verlag Berlin Heidelberg, 1992.

[4] K. Bandara, F. Cirak, G. Of, O. Steinbach, J. Zapletal. Boundary element based multiresolution shape optimisation in electrostatics. *Journal of Computational Physics, 297, pp. 584-598,Elsevier*, 2015.

[5] K. Welker, M. Siebenborn. Algorithmic aspects of multigrid methods for optimization in shape spaces. 2017, arXiv: 1611.05272v3.

[6] J. M. Lee. *Introduction to Smooth Manifolds,, Second Edition*. Springer, Graduate Texts in Mathematics, 2013.

[7] H. P. Langtangen, A. Logg. *Solving PDEs in Python - The FEniCS Tutorial Volume I*. Springer, 2017.

[8] M. S. Alnæs, A. Logg. *UFL Specification and User Manual 0.3*. www.fenics.org, 2010.

[9] M. J. D. Powell. Algorithms for nonlinear constraints that use lagrangian functions. *Mathematical Programming 14*, 1976.

[10] K. Burg, H. Haf, F. Wille, A. Meister. *Partielle Differentialgleichungen und funktionalanalytische Grundlagen, 5. Auflage*. Vieweg +Teubner Verlag, Springer Fachmedien, 2010.

[11] B. Zhong P.A. Sherar, C.P.Thompson, B. Xu. An optimization method based on b-spline shape functions & the knot insertion algorithm. *Proceedings of the World Congress on Engineering*, II, 2007.

[12] Q. Wang, J. Wang, J. Chen, S. Luo, J. Sun. Aerodynamic shape optimized design for wind turbine blade using new airfoil series. *Journal of Mechanical Science and Technology, Vol. 29, Issue 7, pp. 2871-2882*, 2015.

[13] S. Schmidt. Weak and strong form shape hessians and their automatic generation. 2018, SIAM J. Sci. Comput., Vol. 40, No.2, pp. C210-C233.

[14] S. Schmidt, C. Ilic, V. Schulz, N. R. Gauger. Three-dimensional large-scale aerodynamic shape optimization based on shape calculus. *AIAA J., 51, pp. 2615-2627*, 2013.

[15] Volker Schulz. A riemannian view on shape optimization. *Foundations of computational Mathematics*, 14:483-501, 2014.

[16] B. Schweizer. *Partielle Differentialgleichungen - Eine anwendungsorientierte Einführung*. Springer Spektrum, 2013.

[17] Volker Schulz, Martin Siebenborn. Computational comparison of surface metrics for pde constrained shape optimization. *Comput. Methods Appl. Math 2016*, 2016.

[18] Kevin Sturm. *On shape optimization with non-linear partial differential equations*. PhD thesis, Technische Universität Berlin, 2015.

[19] L. Nazareth, T. G. Kolda, D. P. O'Leary. BFGS with update skipping and varying memory. *SIAM J. Optim.*, 8., No. 4, 1998.

[20] T. Tartar. An introduction to Sobolev and interpolation spaces. 2007, Springer, Berlin, Heidelberg.

[21] W. Arendt, K. Urban. *Partielle Differenzialgleichungen - Eine Einführung in analytische und numerische Methoden*. Spektrum Akademischer Verlag Heidelberg, 2010.

[22] K. Welker. Suitable Spaces for Shape Optimization. 2017, arXiv: 1702.07579v2.

[23] Kathrin Welker. *Efficient PDE Constrained Shape Optimization in Shape Spaces*. PhD thesis, Universität Trier, 2016.

[24] Volker Schulz, Martin Siebenborn, Kathrin Welker. Towards a lagrange-newton approach for constrained shape optimization. *arXiv: 1405.3266v2*, 2014.

[25] Volker Schulz, Martin Siebenborn, Kathrin Welker. Pde constrained shape optimization as optimization on shape manifolds. *Geometric Science of Information, Lecture Notes in Computer Science*, 9389:pp. 499–508, 2015.

[26] Volker Schulz, Martin Siebenborn, Kathrin Welker. Structured inverse modeling in parabolic diffusion problems. 2015.

[27] Volker Schulz, Martin Siebenborn, Kathrin Welker. Efficient pde constrained shape optimization based on steklov-poincaré-type metrics. *SIAM J. OPTIM.*, Vol. 26, No. 4, pp. 2800-2819, 2016.

[28] Jorge Nocedal, Stephen J. Wright. *Numerical Optimization, Second Edition.* Springer, 2006.

[29] M. C. Delfour, J. P. Zolésio. *Shapes and Geometries: Metrics, Analysis, Differential Calculus, and Optimization, 2nd ed.* SIAM Advances in Design and Control, 2011.