1 Implementierung in Python mit FEniCS

1.1 ???

mache 2 große sections bib und main, dann bei bib untersections gitter, lösen von pdes und b
Nachdem wir in vorigen Kapiteln den theoretischen Hintergrund der Formoptimierung und gradientenbasierte Verfahren, wie dem L-BFGS-Verfahren, gelegt haben, möchten wir in diesem Abschnitt die Implementierung des Algorithmus cite in Python 3.5 Versicherung, dass es nur unter dieser Version läuft 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. Die Implementierung in Python besteht im wesentlichen 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 L-BFGS-Algorithmus mit den damit verbundenen Objekten gebündelt sind. Die Berechnungen auf den Formen und das Lösen der PDEs wird mittels FEniCS geschehen. FEniCS ist eine https://fenicsproject.org/ frei zugängliche Programmierung, welche 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 [5]. Diese nutzen wir, um 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.

subsection gitter?

Gitterdateien erzeugen wir mit Hilfe des offen zugänglichen Programms Gmsh 3.0.6 http://gmsh.info/. Hierbei muss man zunächst eine .geo Datei geschrieben werden. Wir zeigen dies am Beispiel eines kleinen Kreises. Zunächst setzen wir die für unser Gitter relevanten Punkte in ein 3-dimensionales Koordinatensystem

```
1 Point(1) = {0.0, 0.0, 0.0, 1.0};
2 Point(2) = {1.0, 0.0, 0.0, 1.0};
3 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};

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 mit dem Kommando

```
gmsh mesh_smallercircle.geo -2 -clscale 0.025
```

in eine .msh Datei konvertiert. Dabei ist mesh_smallercircle.geo der Name der gespeicherten .geo-Datei, -2 die Dimension des erzeugten Gitters und 0.025 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.
neuer Abschnitt?
Nun besitzen wir die nötigen Gitterdateien, um auf diesen Formoptimierung zu betreiben. Wir stellen kurz vor, mit welchen Objekten und Funktionen wir mit diesen umgehen. Eines der beiden zentralen Objekte des Optimierungsprogramms ist die sogenannte *MeshData-Klasse*.

```
1  class MeshData:
2
3      # Objekt mit allen Daten des Gitters
4      def __init__(self, mesh, subdomains, boundaries, ind):
5
6          # FEniCS Mesh
7          self.mesh = mesh
8
9          # FEniCS Subdomains
10         self.subdomains = subdomains
11
12         # FEniCS Boundaries
13         self.boundaries = boundaries
14
15         # Indizes der Knotenpunkte mit Traeger nicht
16         # am inneren Rand
17         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 als 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 zurückgibt, welche den Nummern der Physical Surface bzw. Physical Line in der .geo-Datei entsprechen. Diese Initialierungen 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.

neuer Abschnitt lösen von PDE und berechnungen, falls oben noch irgendwas dazugehören ka

Nun besitzen wir Gitterobjekte, auf welchen wir Berechnungen durchführen können. Der Hauptanteil der Berechnungen besteht in dem Lösen von partiellen Differentialgleichungen. Die Gleichungen, welche wir für die Formoptimierung lösen müssen, haben wir in zitat kapitel vorher eingeführt. Dabei handelt es sich um die Poisson-Zustandsgleichung ref, die zugehörige adjungierte Gleichung ref, und die lineare Elastizitätsgleichung ref, 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 im wesentlichen analog vorgegangen, hierbei verweisen wir auf unseren Quellcode zitat?. 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 aus ref lame param bekannten 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, unter 1.1. 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 [11], 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 beispeilsweise mit

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

initialisiert. Um die lineare Elastizitätsgleichung zu lösen, müssen wir zunächst den dazugehörigen 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 etwa [4]. 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. Wir initialisieren diese mittels

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

sage, dass damit nicht so einfach gearbeitet werden kann, vielleicht das vorweg nehmen, oder

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 ref bilinform bekannte Bilinearform, und shape_deriv die in ref shape deriv angegebene Formableitung. 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 Lamé-Parameter mu_elas. eigentlich lame oder? passt eig. Hier kommt die stärke von FEniCS zur Geltung, nähmlich die Verwendung der

eingangs erwähnten *Unified Form Language*. Die hier Initialisierten Objekte sind exakt die Objekte, welche in der schwachen Formulierung der Gleichung ref lin elas auftauchen, was der mathematischen Schreibweise sehr nahe steht, und somit die Lesbarkeit 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 programmiertechnischen Details der Objekte der UFL den Rahmen dieser Arbeit sprengen würde, verweisen wir für den interessierten Leser auf [4] und [5].
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

$$D\mathcal{J}(\Omega_2)[\varphi_i] \mathrel{\hat{=}} \text{F\_elas.get\_local()[i]} \tag{1.1}$$

interpretieren, wobei φ_i polynomielle Basisfunktionen vom Grad 1 auf dem Gitter sind, welches Ω_2 repräsentiert. Auf diese Weise lässt sich $D\mathcal{J}(\Omega_2)[\cdot]$ über F_elas als skalarwertige Funktion auf dem initialisierten Gitter auffassen,

wobei die jeweiligen Werte des Knoten mit Index i genau den Ableitungen $D\mathcal{J}(\Omega_2)[\varphi_i]$ 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 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 sieht scheisse aus

$$d_{shp}(\partial\Omega_1, \partial\Omega_2) := \int_{\partial\Omega_2} \min_{y\in\partial\Omega_1} ||x-y|| dx \tag{1.2}$$

messen werden. Diese Distanzfunktion wird bei uns lediglich als Ersatz für den Abstand zweier Formen im Shape-space verwendet, welchen man mittels Geodätischer definieren kann. Um jedoch 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} ist nicht symmetrisch, 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 List 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 1.1 ähnliche Assemblierung des Integrals statt, welche uns den Abstandwert als Return liefert.
vllt noch ein bild mit dem Abstandswerten als Fenics funktion mit Wert drunter
vllt noch restliche auf gitter rechnende Funktionen nennen neuer abschnitt
Die von uns bisher eingeführten Objekte würde schon ausreichen, um ein Verfahren auf Basis des Gradientenabstiegs zu programieren. Wir möchten jedoch einen Schritt weiter gehen, und den L-BFGS-Algorithmus implementieren. Das Hauptproblem bei dem Verfahren besteht darin, sich eine Möglichkeit zu überlegen, mit dessen Hilfe man die Gradienten- und Deformationsfelder speichern und updaten kann, da wir ja nur eine *limited memory* besitzen. Hierzu entwerfen wir die zweite wichtige Klasse in unserem Program, die sogenannte bfgs_memory-Klasse. Diese ist wie folgt aufgebaut, wobei wir im Program 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 Zahlen 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 den Schritt 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 [18], 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 des 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 Werte, sobald die Anzahl length an gespeicherten Einträgen überschritten wird, löscht, und alle anderen dem Index nach um 1 aufrückt.
vllt ein einfaches Diagram

```
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 ref 2 loop 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 wird zudem der 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 lange Codesequenzen sind, 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 Hesseoperator 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 Gradientenfeld -memory.gradient[0] 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. D.h. die Memory muss auf dem aktuellst möglichen Stand sein. bfgsstep

Literatur

[1] M. Genzen, A. Staab, Prof. E. Emmrich. Sobolew-Slobodeckij-Räume - die Theorie der gebrochenen Sobolew-Räume, Technische Universität Berlin. 2014.

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

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

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

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

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

[7] 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.

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

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

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

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

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

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

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

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

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

[17] 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.

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

[19] 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.

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

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