

# 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  
InitialData ordner anlegen?? Hadamard abstand abwandlung erklären erwähne die Impleme

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};
```

## 1 Implementierung in Python mit FEniCS

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

```
1 Line(1) = {1, 2};
2 Line(2) = {2, 4};
3 Line(3) = {4, 3};
4 Line(4) = {3, 1};
5
6 Circle(5) = {5, 6, 7};
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.

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

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

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

## 1 Implementierung in Python mit FEniCS

```
7
8 Physical Surface(1) = {1};
9 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
```

## 1 Implementierung in Python mit FEniCS

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 Initialisierungen geschehen über die Befehle

```
1 mesh      = Mesh(path_meshFile + ".xml")
2 subdomains = MeshFunction("size_t", mesh,
3                             path_meshFile + "_physical_region.xml")
4 boundaries = MeshFunction("size_t", mesh,
5                             path_meshFile + "_facet_region.xml")
6 ind        = __get_index_not_interior_boundary(mesh, subdomains,
7                                                  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

## 1 Implementierung in Python mit FEniCS

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

```
1 solve_state(meshData, fValues)
2 solve_adjoint(meshData, y, z)
3 solve_linelas(meshData, p, y, z, fValues,
4               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 [12], 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

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

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

## 1 Implementierung in Python mit FEniCS

```
1 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

```
1 u_out = Constant((0.0, 0.0))
2 bcs = [DirichletBC(V, u_out,
3               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

```
1 U = TrialFunction(V)
2 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:

```
1 LHS = bilin_a(meshData, U, v, mu_elas)
2 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:

```
1 def bilin_a(meshData, U, V, mu_elas):
2
3     dx = Measure("dx", domain=meshData.mesh,
4               subdomain_data=meshData.subdomains)
5
6     epsilon_V = sym(nabla_grad(V))
7     sigma_U = 2.0*mu_elas*sym(nabla_grad(U))
8
9     a = inner(sigma_U, epsilon_V)*dx('everywhere')
10    value = assemble(a)
11
12    return value
```

## 1 Implementierung in Python mit FEniCS

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, nämlich 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.

```
1 if (zeroed): F_elas[meshData.indNotIntBoundary] = 0.0
2
3 for bc in bcs:
4     bc.apply(LHS)
5     bc.apply(F_elas)
```

Das Lösen der nun aufgebauten Gleichung erfolgt mit dem Befehl.

```
1 U = Function(V, name="deformation_vec")
2 solve(LHS, U.vector(), F_elas)
3
4 # Rückgabe des Deformationsvektorfeldes U und der
   Abbruchbedingung
5 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

$$D\mathcal{J}(\Omega_2)[\varphi_i] \triangleq F\_elas.get\_local()[i] \quad (1.1)$$

## 1 Implementierung in Python mit FEniCS

interpretieren, wobei  $\varphi_i$  polynomielle Basisfunktionen vom Grad 1 auf dem Gitter sind, welches  $\Omega_2$  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  $\varphi_i$  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

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

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 Abstandsfunktion  $d_{shp}$  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)
```



## 1 Implementierung in Python mit FEniCS

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

## 1 Implementierung in Python mit FEniCS

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 [19], Abschnitt 4. Anschließend lässt sich bei Bedarf mit diesen Daten eine neue FEniCS-Funktion auf einem neuen Gitter initialisieren.

```
1 def initialize_grad(self, meshData, i):
2
3     if isinstance(meshData, MeshData): pass
4     else: raise SystemExit("initialize_grad benoetigt Objekt der
      MeshData-Klasse als Input!")
5
6     V = VectorFunctionSpace(meshData.mesh, "P", 1, dim=2)
7     f = Function(V)
8     f.vector()[:] = self.gradient[i]
9     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 Werte, sobald die Anzahl `length` an gespeicherten Einträgen überschritten wird, löscht, und alle anderen dem Index nach um 1 aufrückt.

vllt ein einfaches Diagramm

```
1 def update_grad(self, upd_grad):
2
3     for i in range(self.length-1):
```

## 1 Implementierung in Python mit FEniCS

```
4     self.gradient[-(i+1)] = self.gradient[-(i+2)]
5
6     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 approximierten Hesseoperator ausgewertet wird. So gibt die Funktion `bfgs_step` eine vektorwertige FEniCS-Funktion zurück, welche als Deformationsvektorfeld dienen kann, und mit

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

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. Jetzt besitzen wir das gesamte Rüstzeug, um in der Hauptdatei die Bausteine miteinander zu vernetzen.

neues großes Kapitel; Maindatei; zuerst Einstellbarkeit und Parameter; dann Verfahren und a  
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 Programs als `integer`-Parameter mit dem Befehl

## 1 Implementierung in Python mit FEniCS

```
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. Das Startgitter wäre in diesem Fall das eines fein aufgelösten kleinen Kreises, der zweite Eintrag das Zielgitter eines fein aufgelösten, nierenartigen Form. **Bild einfügen?**

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 `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 1.1 eingeführte Wert der Norm `nrm_f_elas` die einstellbare Toleranz `tol_shopt` unterschreitet.

Als Parameter des Ausgangsproblems lassen sich außerdem noch die Werte der Zustandsgleichung **referenz** innerhalb und außerhalb der Form, `f_1` und `f_2` einstellen, sowie auch der Werte des Parameters der Périmeter-Regularisierung `nu`. Letzterer sollte problemabhängig nicht zu groß gewählt werden, um eine Optimalität des Zielgitters nicht stark zu verfälschen. Hinzu kommen zudem die minimalen und maximalen Werte der Lamé-Parameter `mu_min` und `mu_max`.

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 Line-Search durch Backtracking einschalten, indem man den Parameter `Linesearch`

## 1 Implementierung in Python mit FEniCS

auf `True` setzt. Die weiteren Parameter und die Implementierung der Line-Search werden weiter unten bei [Referenz](#), und in der folgenden alle Parameter und Einstellungen umfassenden Tabelle näher erklärt

[mache hier eine Tabelle wie in einem Nutzerhandbuch zu allen Parametern in der Auswahl jetzt abschnitt zur genauen Implementierung; Schema des Programs; line-search; output](#)

Im Folgenden möchten wir den genauen Ablauf des Programs in Python erläutern. Das Program läuft nach dem Schema

[mit ktikz ein Flussdiagramm wie bei bayesian inference machen](#) ab.

Das Program beginnt mit der in [subchapter meshes](#) erklärten Initialisierung der Meshes. Je nach Einstellung findet die oben erklärte Perturbation des Startgitters statt. Weiterhin werden die lokal variierenden Lamé-Parameter, sowie die Zustandsfunktion im Zielgitter berechnet. Dies findet im Program unter der Überschrift *Targetdata Calculation* statt. Es folgt der Einstieg in den Formoptimierungsalgorithmus, wobei unter *Interpolation* 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 Program unter dem gleichnamigen Schritt zu finden ist. Falls das Gradientenverfahren ausgewählt ist, wird zusätzlich als Deformation der negative Gradient initialisiert.

Nun findet im Program 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 [referenz von LineSearch](#)

Ein programmiertechnisch etwas aufwendiger Punkt ist die Berechnung der Curvature Condition. In unserem Setting ist diese errechnet durch [transporte richtig? Vllt noch die](#)

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

Dies vermeidet die Lösung einer weiteren linearen Elastizitätsgleichung. Nichts desto trotz [rechtschreibung?](#) müssen wir zur Berechnung von  $D\mathcal{J}(\Omega_{k+1})[\mathcal{T}_{k+1}(S_k)]$  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 Program 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

## 1 Implementierung in Python mit FEniCS

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 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 **ref** nicht erfüllt ist, verwenden wir das Verfahren, die Memory nicht einem Update zu unterziehen, d.h. es wurde lediglich ein Schritt mit schon vorhandenen Daten berechnet. Dieses Skipping findet im Program unter dem Abschnitt *Non Update Step* statt. Da der Update sowieso einen Schritt verzögert stattfindet, wird dieser einfach ausgelassen, und das Program läuft weiter. Diese Strategie lässt sich mit der Einstellung des Parameter `curv_break = True` ausschalten, woraufhin der Algorithmus bei Verletzung der Curvature Condition aussteigt.

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 `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.

**ARMIO? ist die Initialisierung wirklich interessant? Name counterer?**

```
1 if (Linesearch):
2     scale_parameter = start_scale
3
4     zero_function = Function(VectorFunctionSpace(MeshData.mesh,
5                                                    "P", 1, dim=2))
6     current_value = bib.targetfunction(MeshData, zero_function,
7                                         y_z, f_values, nu)
8     S.vector()[:] = start_scale * S.vector()
9     current_deriv = bib.shape_deriv(MeshData, p, y, z,
10                                     f_values, nu, S)
11
12     counterer = 0
```

Der Counter `counterer` zählt dabei die Anzahl der Reskalierungen der Deformation. Das eigentliche Backtracking findet in der gleich folgenden Schleife

## *1 Implementierung in Python mit FEniCS*

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 Meldung 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.

## 1 Implementierung in Python mit FEniCS

```

1 while(bib.targetfunction(MeshData, S, y_z, f_values, nu)
2     >= current_value):
3
4     scale_parameter = shrinkage * scale_parameter
5     S.vector()[:] = shrinkage * S.vector()
6
7     counterer = counterer + 1
8
9     if(counterer >= 20):
10        print("Had to break, restarting L-BFGS!")
11
12        bfgs_memory.gradient = np.zeros([memory_length, 2 *
13                                         MeshData.mesh.num_vertices()])
14        bfgs_memory.deformation = np.zeros([memory_length, 2 *
15                                             MeshData.mesh.num_vertices()])
16        bfgs_memory.step_nr = 0
17        Resetcounter = Resetcounter + 1
18        S.vector()[:] = np.zeros(2*MeshData.mesh.
19                                num_vertices())
20        break
21
22        if(counterer < 20):
23            Resetcounter = 0
24
25        last_defo = S.vector().get_local()
26
27 if(Resetcounter >= 2):
28     print("Reboot didn't help, quitting L-BFGS optimization!")
29     break

```

Es ist außerdem möglich, statt der einfachen Backtracking Bedingung, die sogenannte *Armijo-Bedingung* mit Hilfe eines Backtracking-Verfahrens zu implementieren. Dies würde mit einer Schleife der Form

```

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

```

funktionieren. Diese 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 [?], 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 [ref zu anderem kapitel](#) zur Berechnung verwendet, die Differenz der Gradienten



## 1 Implementierung in Python mit FEniCS

ten heißt das ding pk bei uns? sicher defo wie in referenz bfgs verfahren definiert zu ersetzen. 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 Zwei-Schleifen-Algorithmus `ref` verwenden, um die Inverse  $B_k^{-1}$  zu berechnen. Weiterhin führt dieses Verfahren zu einem sogenannten *Blow-Up* des Hesseoperators, weshalb das Verfahren dann neugestartet 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.

Abschließend kommen wir im Kapitel zur Implementierung zum Output des Programs. Zu Beginn wird bei Aufruf des Programs ein Outputordner im Verzeichnis des Programs mit Namen `Output` erzeugt, falls dieser noch nicht vorhanden ist. Dies geschieht mit der von uns implementierten Funktion `create_outputfolder()`. In diesen Ordner wird für jedes Aufrufen des Programs, sprich für jedes Optimierungsproblem, ein eigener Unterordner mit dem Namen erstellt, welches nach dem exaktem jetzigem 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.

Gradientenvektorfeld `output` ist auskommentiert, weg machen? 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 Program *Paraview* `vllt version? oder bsp` visualisiert werden können. Zusätzlich wird bei erfolgreichem Durchlaufen der Optimierung ein Konvergenzplot gespeichert, welcher `vllt noch ein bisschen aufschönen` die Abbildungen

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

zeigt, wobei die Norm wie in 1.1 erklärt berechnet wird. Um eine Analyse der `make noch Parameterauswahl in cvd kenntlich, d.h. Pertub und linesearch` 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 Programs auch für den Benutzer in der Konsole geprintet werden. Schematisch sieht dies wie folgt aus:

## 1 Implementierung in Python mit FEniCS

Iteration    $||f_{\text{elas}}||_{L2}$     $J = j + j_{\text{reg}}$     $||U||_{L2}$    Curv. Cond.   Meshdistance

Falls das Gradientenverfahren ausgewählt wurde, so wird die Spalte mit der Curvature Condition weggelassen. Diese Datei lässt sich im Rahmen des Post-processings 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 *.cvd*-Datei wird trotzdem erstellt. Im nun folgenden und letzten Kapitel werden wir solche Analysen für einige von uns ausgewählte Routinen vorstellen. sollte ich noch implementieren und checke, ob das nicht lohnenswert ist.

## LITERATUR

### 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](http://www.fenics.org), 2010.
- [6] M. J. D. Powell. Algorithms for nonlinear constraints that use lagrangian functions. *Mathematical Programming* 14, 1976.
- [7] K. Burg, H. Haf, F. Wille, A. Meister. *Partielle Differentialgleichungen und funktionalanalytische Grundlagen, 5. Auflage*. Vieweg +Teubner Verlag, Springer Fachmedien, 2010.
- [8] 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.
- [9] S. Schmidt. Weak and strong form shape hessians and their automatic generation. 2018, SIAM J. Sci. Comput., Vol. 40, No.2, pp. C210-C233.
- [10] Volker Schulz. A riemannian view on shape optimization. *Foundations of computational Mathematics*, 14:483-501, 2014.
- [11] B. Schweizer. *Partielle Differentialgleichungen - Eine anwendungsorientierte Einführung*. Springer Spektrum, 2013.
- [12] Volker Schulz, Martin Siebenborn. Computational comparison of surface metrics for pde constrained shape optimization. *Comput. Methods Appl. Math* 2016, 2016.

## LITERATUR

- [13] Kevin Sturm. *On shape optimization with non-linear partial differential equations*. PhD thesis, Technische Universität Berlin, 2015.
- [14] W. Arendt, K. Urban. *Partielle Differenzialgleichungen - Eine Einführung in analytische und numerische Methoden*. Spektrum Akademischer Verlag Heidelberg, 2010.
- [15] K. Welker. Suitable Spaces for Shape Optimization. 2017, arXiv: 1702.07579v2.
- [16] Kathrin Welker. *Efficient PDE Constrained Shape Optimization in Shape Spaces*. PhD thesis, Universität Trier, 2016.
- [17] Volker Schulz, Martin Siebenborn, Kathrin Welker. Towards a lagrange-newton approach for constrained shape optimization. *arXiv: 1405.3266v2*, 2014.
- [18] 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.
- [19] Volker Schulz, Martin Siebenborn, Kathrin Welker. Structured inverse modeling in parabolic diffusion problems. 2015.
- [20] 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.
- [21] Jorge Nocedal, Stephen J. Wright. *Numerical Optimization, Second Edition*. Springer, 2006.
- [22] M. C. Delfour, J. P. Zolésio. *Shapes and Geometries: Metrics, Analysis, Differential Calculus, and Optimization, 2nd ed.* SIAM Advances in Design and Control, 2011.