Programmier-Praktikum WS2020/2021

Shape optimization with FEniCS

Gemeinsame Dokumentation

Dozent: Dr. Martin Lenz

Bonn, am 2. Februar 2021

1 Modellproblem „Brücke“

Ein weiteres anschauliches Modellproblem mit Dirichlet- und Neumannrandwerten zur Formoptimierung ergibt sich durch die Konstruktion einer Brücke, wie in der folgenden Skizze dargestellt wird:

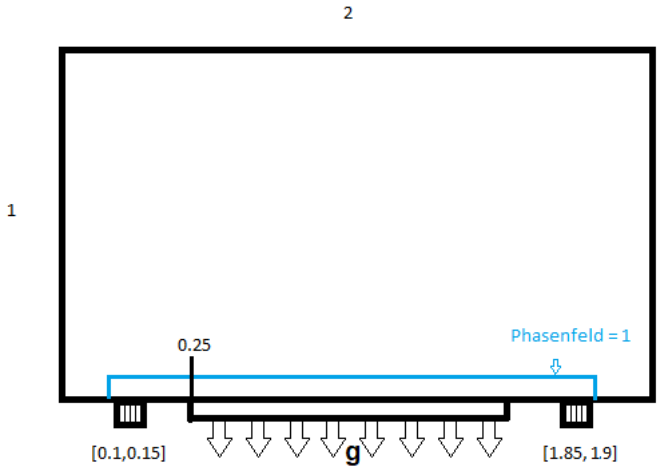

Abbildung 1: Brücke

Für brauchbare Ergebnisse wählen wir dazu ein Gitter, das mindestens die Größe 200 x 100 hat, und definieren dazu:

$$\lambda = \mu = 5, g = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \quad \text{sowie}$$
$$\varepsilon = 0.01, \delta = 0.01, \quad \alpha = 0.1, \quad \beta = 5$$
$$\text{(void)} \qquad \text{(surf)} \quad \text{(volume)}$$

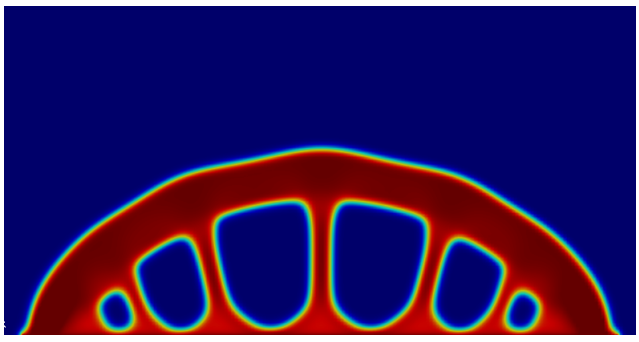

Abbildung 2: Lösungsbild Brücke

2 Formoptimierung mit parametrisierten Deformationen

Wie beschreibe ich eine Form mit scharfen Grenzen?

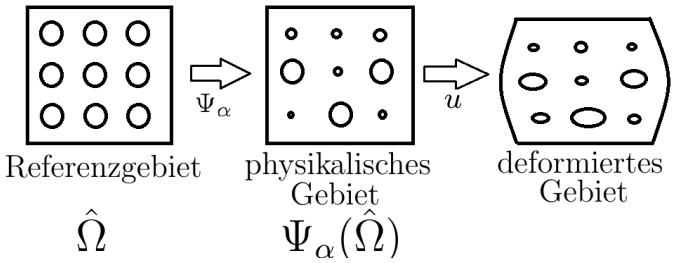

Abbildung 3: Parametrisierung

Da in der Realität lediglich die Werte -1 und 1 für unser Phasenfeld sinnvoll sind, wir aber durch Stetigkeit immer „weiche“ Übergänge erhalten, ergibt sich die Frage, wie wir eine Form mit scharfen Grenzen beschreiben können. Dazu stellt die Parametrisierung eines Gebietes durch ein Referenzgebiet eine sinnvolle Möglichkeit dar.
Wir betrachten dazu nun das oben skizzierte **Beispiel:**
Unser physikalisches Gebiet besteht dabei aus einem Quadrat mit Löchern von variablem Radius - dabei sind α_i die relativen Radien zu den ursprünglichen Radien. Wir müssen nun die folgenden Schritte genauer verstehen:

$$\underset{\text{(Control)}}{\alpha} \overset{①}{\mapsto} \Psi_\alpha \overset{②}{\mapsto} u \overset{③}{\mapsto} \underset{\text{(Red.Funct.)}}{\mathcal{J}} + \overset{④}{\text{Dolfin-Adjoint}}$$

⓪: Zunächst müssen wir unser **Referenzgebiet $\hat{\Omega}$/unser Gitter** definieren bzw. implementieren. Dazu verwenden wir das Paket mshr und die Mengenoperation Vereinigung (+) bzw. Mengendifferenz (-), sowie die Teilfunktionen mshr.Rectangle und mshr.Circle.
→ anschließend verwenden wir generate_mesh und create_overloaded_object(__), wie im folgenden Beispielcode beschrieben wird:

```
geometry = mshr.Rectangle(Point((0,0)), Point((a,b))) - mshr.Circle(Point(c,d),radius)
mesh = create_overloaded_object(mshr.generate_mesh(geometry, n))
```

Abbildung 4: generate Mesh

①: **Definiere** $\Psi_\alpha(\hat{x})$ zunächst nur für $\hat{x} \in \partial\hat{\Omega}$ durch:

$$\Psi_\alpha(\hat{x}) = \begin{cases} \hat{x}, & \hat{x} \in \hat{\Gamma}_{\text{ext}} \\ \alpha_i(\hat{x} - \hat{c}_i) + \hat{c}_i, & \hat{x} \in \hat{\Gamma}_i \end{cases}$$

Dabei ist $\hat{\Gamma}_i$ der Rand eines Loches und Γ_{ext}, wie zu erwarten, der äußere Rand unseres Referenz- sowie physikalischen Gebietes. Hierbei ergibt sich natürlich die Frage, wie Ψ_α im Inneren von $\hat{\Omega}$ definiert sein soll.
Dazu lösen wir die Laplace-Gleichung $\Delta\Psi_\alpha = 0$ mit Dirichlet Randwerten. Das ist sinnvoll, da die Laplace-Gleichung als elliptische Gleichung einen Glättungsprozess vollzieht. (schwache Form: $\int_{\hat{\Omega}} \nabla\Psi_\alpha : \nabla\varphi = 0$).
Dies ist lediglich unter Umständen problematisch, wenn Kreise zu sehr aufgeblasen werden (ggf. remeshen)
Code für FEniCS:

```
alpha_i = Constant(...)
psi = project(alpha_i * (x_hat - center_i) + center_i,V)
DirichletBC(V,psi,...)
```

Abbildung 5: Variante 1

Dieses Verfahren ist durch den Operator project allerdings sehr aufwendig, da project für jedes Loch ein LGS auf dem gesamten Gitter löst, weshalb wir uns nun mit einer zweiten Variante der Definition von Ψ_α beschäftigen wollen. Dazu verwenden wir die Methode assign zusammen mit einer Hilfsfunktion bc:

```
bc = Function(V)
bc.assign(alpha_i * (x_hat - center_i) + center_i)
```

Abbildung 6: Variante 2

(Linearkombination erlaubt, effizient!)
Allerdings führt diese Variante zu einem Problem in ④, wie wir im Folgenden sehen werden.

②: Nun ist es unser Ziel, die **elastische Energie** auf $\Omega = \Psi_\alpha(\hat{\Omega})$ abhängig von unserem Referenzgebiet $\hat{\Omega}$ zu formulieren. Dazu benötigen wir den Transformationssatz sowie grundlegende Überlegungen der wechselseitigen Abhängigkeiten der Ableitungen nach x bzw. $\hat{x}$.
(Dabei verzichten wir in den folgenden Betrachtungen auf das Randintegral, da dieses unverändert bleibt.)

$$E = \int_\Omega \frac{1}{2}\sigma(u) : \varepsilon(u)dx + \dots$$

$$\sigma(u) = \lambda \text{tr}(\varepsilon(u))\mathbb{1} + 2\mu\varepsilon(u)$$

$$\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$$

Wir wollen nun also den Gradienten von u abhängig von den Ableitungen nach $\hat{x}$ statt nach x zu formulieren, dazu folgende Betrachtungen:

Ableitungen:

Da

$$x = \Psi(\hat{x}) \qquad \text{und} \qquad \nabla_{\hat{x}} u(\Psi(\hat{x})) = \nabla_x u(\Psi(\hat{x})) \nabla_{\hat{x}} \Psi(\hat{x})$$

ist also

$$\nabla u = \nabla_x u(x) = \nabla_x u(\Psi(\hat{x})) = \nabla_{\hat{x}} u(\Psi(\hat{x})(\nabla_{\hat{x}} \Psi(\hat{x}))^{-1}$$

$$\boxed{\Rightarrow \nabla u = \nabla_{\hat{x}} u(\Psi(\hat{x})(\nabla_{\hat{x}} \Psi(\hat{x}))^{-1}}$$

Integral: (Transformationssatz)

$$\int_\Omega f(x)dx = \int_{\Psi(\hat{\Omega})} f(x)dx \overset{\text{Trf.}}{=} \int_{\hat{\Omega}} f(\Psi(\hat{x}))|\det(D\Psi(\hat{x}))|d\hat{x}$$

Ohne Beträge, falls Ψ_α orientierungserhaltend.
Compliance unverändert. Γ_{ext} fest

③: Wir bestimmen nun unser **Zielfunktional**:
$\mathcal{J}$ =Compliance+Volume

$$\text{Volume} = \int_\Omega 1dx = \int_{\hat{\Omega}} \det(D\Psi(\hat{x}))d\hat{x}$$

④: **Dolfin_Adjoint**
Wir initialisieren unsere Konstanten α_i mit einem beliebigen Startwert. Hier wählen wir dazu $\alpha_i = 1$, da dies äquivalent ist zu gleich bleibenden Radien:

$$\alpha_i = \text{Constant}(1.0)$$

Nun deklarieren wir jede dieser Konstanten zur Kontrolle:

$$c_{\alpha_i} = \text{Control}(\alpha_i)$$

und definieren die Kontrollvariablen als die Liste ebendieser:

$$\text{controls} = [c_{\alpha_0}, c_{\alpha_1}, \dots]$$

Dieses Verfahren funktioniert allerdings nur mit der naiven 1.Variante, wie wir sie in ② beschrieben haben.

Um genauer zu verstehen, wieso die in ② beschriebene zweite Variante hier zu einem Problem führt, betrachten wir ein Beispiel (siehe Abbildung 7 (Beispielcode) auf S.6):
Wir definieren unser Gitter als UnitSquareMesh mit Parameter 32, sowie Funktionen $a = x, b = -x$ und $c = e \cdot a + (1 - e) \cdot b$
Nun möchten wir das Funktional $\mathcal{J} = c * c * dx$ bzgl. e minimieren, erhalten allerdings

bei der in ② beschriebenen zweiten Implementierungsmöglichkeit eine Fehlermeldung. Nach genauerer Betrachtung des Tapes stellen wir fest, dass das maßgebliche Problem darin besteht, dass die Abhängigkeit von e (in unserem ursprünglichen Beispiel also die Abhängigkeit von den α_i) nicht erkannt wird.

Lösung: führe Instanz der Skalarmultiplikation (smul) ein, die nicht mit in das tape eingeht (Vgl. Code) und die Abhängigkeit der Skalaren nicht vernachlässigt.

2.1 Was tut evaluate_adjoint_component?

Bsp: $\mathcal{J}(f(g(x)))$

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^m \xrightarrow{f} \mathbb{R}^k \xrightarrow{\mathcal{J}} \mathbb{R}$$

Dann ist:

$$D_x\mathcal{J}(f(g(x))) = DJ(f(g(x)))Df(g(x))Dg(x)$$

Die benötigte **Adjungierte** ergibt sich dann durch:

$$\overline{D_x\mathcal{J}(f(g(x)))}^T = \overline{Dg(x)}^T\overline{Df(g(x))}^T\overline{D\mathcal{J}(f(g(x)))}^T$$

$\Rightarrow$ Berechne $\overline{Df(y)}^T z$

$f : \mathbb{R}^m \to \mathbb{R}^k$
input: $y \in \mathbb{R}^m$
idx: Index, falls mehrere inputs
adj_input: $z \in \mathbb{R}^k$
return: $\overline{Df(y)}^T z \in \mathbb{R}^m$

3 Exercise until 10th February 2021

Implement the shape optimization algorithm with parametrized boundary:

- Create a square mesh with a regular lattice of circular holes.
- Compute a deformation that changes the radii of the holes depending on a list of scalar parameters (using project() in the Dirichlet boundary conditions).
- Compute the elastic energy on the deformed mesh by an integral over the unformed reference mesh.
- Implement the shape optimization algorithm for the carrier plate problem using dolfin-adjoint.

Additional exercises (optional):

- Experiment with the radius of the holes in the reference mesh. How large / small can you allow the deformed holes to become?
- Replace the project()-call in the Dirichlet boundary by a linear combination. Add the missing overloaded methods to dolfin-adjoint.

```python
from fenics import*
from fenics_adjoint import *

from pyadjoint import Block
from pyadjoint.overloaded_function import overload_function

def smul (fac, fun):
    res = Function(fun.function_space())

    res.assign(fac*fun, annotate=False)
    return res

def cdiff(a,b):
    return Constant(a-b)

backend_smul = smul

backend_cdiff = cdiff

class ScalarMultiplicationBlock(Block):-

class ConstantDifferenceBlock(Block):
    def __init__(self,a,b, **kwargs):
        super(ConstantDifferenceBlock,self).__init__()
        self.kwargs = kwargs
        self.add_dependency(a)
        self.add_dependency(b)
    def __str__(self):
        return "ConstantDifferenceBlock"
    def recompute_component(self, inputs, block_variable, idx, prepared):
        return backend_cdiff(inputs[0],inputs[1])
    def evaluate_adj_component(self, inputs, adj_inputs, block_variable, idx, prepared=None):
        inp = adj_inputs[0]
        if idx == 0:
            return idx
        else:
            return -inp


smul = overload_function(smul, ScalarMultiplicationBlock)

cdiff = overload_function(cdiff, ConstantDifferenceBlock)

mesh = UnitSquareMesh(32,32)
V = FunctionSpace(mesh,"CG",1)

x = project(SpatialCoordinate(mesh)[0],V)

a = Function(V, name="a")
a.assign(x)
b = Function(V, name="b")
b.assign(-x)

c = Function(V, name="c")

e = Constant(0.75, name="e")

#c = project(e*a+(1-e)*b,V)                          #a) correct but slow (var1)
#c.assign(e*a + Constant(1-e)*b)                     #b) correct for eval
c.assign(smul(e,a) + smul(cdiff(Constant(1),e),b))  #c) correct and the smart way!

J = c * c * dx

print(assemble(J)) #should be 1/12, works in all cases

#get_working_tape().visualise("tape")

if True:
    #derivative w.r. to scalar: fails in case b)
    Jhat = ReducedFunctional(assemble(J), Control(e))
    Jhat.optimize_tape()
    dir = Constant(0.25)
    taylor_test(Jhat,e,dir)

    #minimization with respect to scalar: fails in case b)
    print(float(minimize(Jhat))) #should be 0.5


else: -
```

Abbildung 7: Beispielcode