Programmierpraktikum der numerischen Simulation: Formoptimierung mit FEniCS

Dozent: Martin Lenz

Skript zum 11.11.2020

Autor: Alexander Becker

Der größte Teil des heutigen Meetings handelte von einer

Einführung in FEnicS

Vieles spielt sich in der sogenannten UFL (*uniform form language*) ab. Diese ist in Python eingebettet und dient dazu, schwache Differentialgleichungen in den Computer zu übertragen, sodass sie verarbeitet und numerisch gelöst werden können.

Ihre Grundlagen haben wir in der Reihenfolge erlernt, wie man auch eine Differentialgleichung in den Computer eingibt. Zuerst muss man festlegen, welche Art von Gitter und Finite-Elemente-Raum für die Lösung herangezogen werden soll. Dann kümmert man sich um die richtige Eingabe der Differentialgleichung mit ihren Randwerten. Nachdem man das System die Gleichung lösen lässt, kann man seine Ergebnisse in eine Datei exportieren und in *ParaView* veranschulichen.

Daher kann dieses Skript nicht nur als eine Zusammenfassung der heutigen Vorlesung, aber auch als eine Anleitung angesehen werden.

0. Importieren

```
from dolfin import *
import ufl
```

1. Gitter

Für die Finite Elemente-Methode braucht man zunächst ein Gitter der einzelnen Punkte. Dafür nehmen wir hier

```
mesh = UnitSquareMesh (64, 64)
```

(Gebiet wo man die Gleichung lösen will, Anzahl Unterteilungen in x- und y-Richtung)

2. Finite-Elemente-Raum

```
V = FunctionSpace (mesh, 'CG', 1)
```

(Definitionsbereich, FE-Typ [hier CG = continous Galerkin, stetige Funktionen mit klassischem Galerkin-Ansatz], Polynomgrad)

3. Randwerte (a-c: Nur zu Dirichlet)

a) Dirichlet-Randfunktion (falls bekannt) festlegen

```
u_D = Constant (0)
```

b) zu betrachtenden Rand einteilen. Hier als Indikatorfunktion (bool)

```
def lrboundary (x, on_boundary):
        return on_boundary and (near (x[0], 0) or near (x[0], 1))
```

Der Parameter `on_boundary` wird erstmal vom System bestimmt. Dann kann man noch spezifizieren, ob x auf einem bestimmten Stück des Randes liegen soll. Hier: ob x auf dem linken oder rechten Rand des Einheitsquadrates liegt. (Äquivalent: dass die 0-te Komponente von x gleich 0 oder 1 ist.)

Analog der zweite, obere, Teil des Randes:

```
def tboundary (x, on_boundary):
        return on_boundary and near (x[1], 1)
```

Auf den Teilen des Randes, die hier nicht eingegeben werden, werden später automatisch vom Compiler natürliche Randwerte festgelegt (d.h. Neumann Null-Randwerte, Ableitung in Richtung „außen“ = Null).

(Das `near` muss hier bloß verwendet werden, da man `double`-Zahlen nicht direkt vergleichen darf.)

c) Dirichlet-Randinformation in eine Variable speichern

```
bc = DirichletBC (V, u_D, lrboundary)
```

d) nach Wunsch noch mehr Randwerte hinzufügen

Falls man für genau nur einen Teil des Randes Neumann-Randwerte benutzen will, muss man noch vor c) eine Klasse definieren.

```
class Top (SubDomain):
        def inside (self, x, on_boundary):
                return tboundary (x, on_boundary)
top = Top ();
```

Wollen Unterklasse von `SubDomain`, da `SubDomain` die (erbbaren) Funktionen hat, um Teile des Gebiets zu markieren.

```
subdomains = MeshFunction ("size_t", mesh, mesh.topology().dim() - 1)
```

Diese Mesh-Funktion lebt auf dem anfangs festgelegten diskreten Gitter `mesh`. Da wir den *Rand* markieren möchten, muss die Funktion in der Dimension des Gitters minus 1 leben. Also schreiben wir `mesh.topology().dim() - 1.`

```
subdomains.set_all (0)
```

Erstmal legen wir die Funktion überall auf 0 fest, dann auf dem gewünschten Randteil (hier genau: den oberen äußeren Kanten des Gitters) auf 1. (-> *Indikatorfunktion*)

```
top.mark (subdomains, 1)
```

Die Instanz `top` hat die Funktion mark von `SubDomain` **geerbt**, die wir hiermit einsetzen, um den oberen Rand zu markieren.

Zu guter Letzt müssen wir noch für später unser Randmaß `ds` ändern, sodass es nur noch auf unseren gewünschten „Neumann-Teil" des Randes misst.

```
ds = Measure ('ds', domain=mesh, subdomain_data=subdomains)
```

(Typ ´ds´: Oberflächenmaß)

4. Ansatz- und Testfunktion erstellen, gegebene Ableitungs-/Randwerte eingeben

```
ut = TrialFunction (V)          (Ansatzfunktion)
phi = TestFunction (V)          (Testfunktion)
f = Constant (-1)               (was Δu sein soll)
g = Constant (-0.2)             (Neumann-Randwert)
```

5. Beide Seiten der Differentialgleichung eingeben

Linke Seite:

```
a = dot (grad(ut), grad(phi)) * dx
```
(Bilinearform, dx = Volumenmaß auf Gitter)

Rechte Seite:

```
L = f * phi * dx + g * phi * ds(1)
```

Wie eben erwähnt: Falls die Neumann-Randwerte nur auf einem bestimmten Teil des Randes berücksichtigt werden sollen, muss man dafür wie oben beschrieben das Standardmaß `ds` ändern. Daher steht hier `ds(1)`. Für überall geltende Neumann-Randwerte muss man nichts machen, nur `ds` hinschreiben.

6. Lösen lassen

```
sol = Function (V, name="solution", bc)
```
(hier wird die Lösung gespeichert)

```
solve (a == L, sol)
```

7. Exportieren in eine Datei

```
file = XDMFFile ("laplace.xdmf")
file.parameters ["functions_share_mesh"] = True
file.write (sol, 0)
```

Ein alternativer Ansatz unseres Elastizitätsproblems war, es durch eine **Minimierung einer Energie** zu lösen. Dies ist im Falle einer nichtlinearen DGL nützlicher.

Hierfür ersetzen wir Schritte 4 – 6 durch:

```
u = Function (V)
energy = ( 1/2 * dot (grad (u), grad (u)) - f * u ) * dx - g * u * ds(1)
deriv = derivative (energy, u)


sol = Function (V, name="solution")
solve (deriv == 0, u, bc)
```

Und entsprechend Schritt 7:

```
file.write (u, 0)
```

Nun ist allerdings unschön, dass der Compiler dieses Problem nicht mehr als **linear** erkennt. Daher löst er es nicht mit dem linearen Verfahren aus letzter Vorlesung, sondern allgemein durch ein Newton-Verfahren. Der Compiler konnte nicht wissen, ob die Energie linear war oder nicht. Auch wenn das Newton-Verfahren für lineare Gleichungen in einem Schritt (bzw. wegen der Computerungenauigkeit in 2 Schritten) konvergiert, hätten wir lieber das lineare.

Also ändern wir/fügen wir hinzu:

`deriv = derivative (energy, u, phi)`	(schwaches) Ableiten in Richtung phi
`deriv2 = ufl.algorithms.replace (deriv, {u: ut})`	Ersetzen von u durch eine Ansatzfunktion ut

Damit enthält `deriv2` die Ansatzfunktion `ut` und Testfunktion `phi`, wodurch eine lineare Form zustandekommt. Getrennt in linke und rechte Seite ist das

`a = lhs (deriv2)`	(extrahiert Teile mit Ansatz- und Testfunktion -> die Bilinearform)
`L = rhs (deriv2)`	(extrahiert Teil mit nur Testfunktion (linearen) und wechselt Vorzeichen, da es auf die andere Seite muss.

Also ändern wir Schritt 6 zurück zu

`sol = Function (V, name="solution", bc)`	(hier wird die Lösung gespeichert)
`solve (a == L, sol)`	

Was man nun noch zusätzlich tun kann, ist **Fehler auszurechnen**. Hierfür denken wir uns selbst eine Funktion u aus und berechnen von ihr den Laplace und z.B. die Dirichlet-Randwerte. Diese Berechnungen geben wir dann in das Programm ein und sehen, wie nah das Programm an unsere ausgedachte Funktion kommt.

Sei $u = 1 + x^2 + 2y^2 \Rightarrow u|_{\partial([0,1]\times[0,1])} = u, \Delta u = 6, \partial_\nu u = 4$, wobei die Normalenableitung für den oberen Rand $[0,1] \times \{1\}$ gemeint ist. Wir geben ein

```
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
```

```
(Mit degree kann man betonen, dass es sich um ein Polynom handelt, und den Grad eingben.)
```

```
f = Constant (-6)            (-Δu)
g = Constant (4)             (Neumann-Randwert).
```

Dann lassen wir die errechneten Lösung `sol` mit der gegebenen Funktion `u_D` vergleichen. (Hier ist `u_D` zwar im Programm als Randwert benutzt, doch wir wissen, dass es die gesamte Lösung ist.)

Am Ende lassen wir mithilfe einer Norm (z.B. der H1-Norm), den Fehler berechnen.

```
print (errornorm (u_D, sol, 'H1'))
```

Hierbei kann man durch Ausprobieren sehen, dass eine Verfeinerung des Gitters meist zur Reduzierung des Fehlers führt. Dafür muss man einfach in die Gitterdefinition eine höhere Zahl für x und y angeben.

ParaView

In eine Datei exportierte Resulatate kann man in ParaView veranschaulichen.

Die wichtigsten grundlegenden Tipps in Listenform:

- Nach dem Öffnen der Datei muss man immer 1x auf „*Apply*“ (linke Seite „Properties“, darin oben) klicken, damit die Daten angezeigt werden.

- Daten in 3D anzeigen lassen: Ganz oben in der Menüleiste: Filters -> Alphabetical -> *Warp by Scalar*. Dann noch direkt über dem Anzeigefenster links auf das „2D“-Symbol klicken und zentrieren (siehe nächster Punkt)

- falls Objekt aus der Sichtweite verloren: Oben in der Mitte „Vier Pfeile nach außen“ drücken -> zentriert Kamera auf Objekt

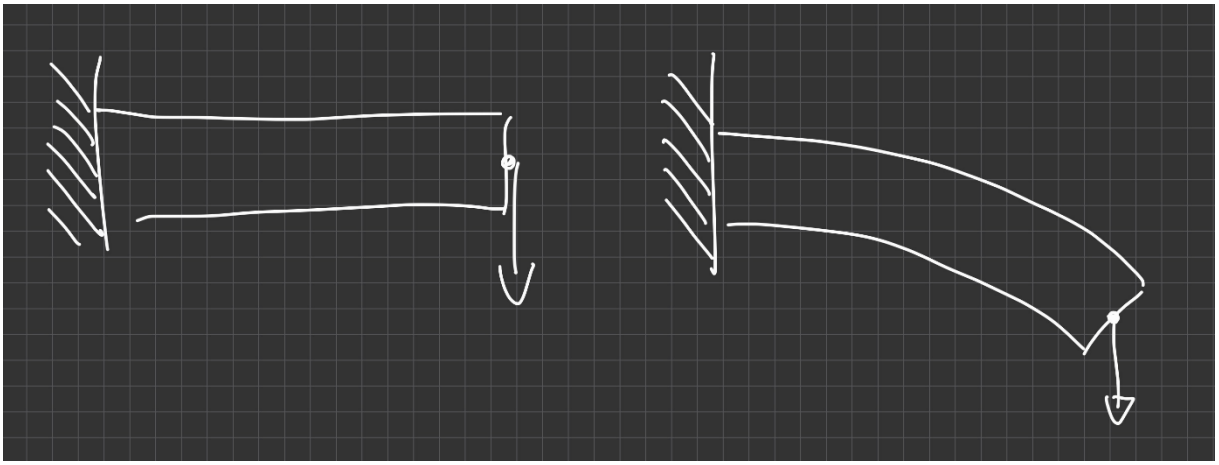
- Achse beim Drehen fixieren: Entsprechenden Buchstaben (*x*,*y* oder *z*) beim Drehen gedrückt halten

- Wertunterschiede amplifizieren: Links: Scale Factor

Elastizitätstheorie

Abschließend noch einige Ergänzungen zur Elastizitätstheorie.

Wir erinnern uns an die Deformationsabbildung φ, die hier von links nach rechts abbildet:



Außerdem definieren wir folgende Größen (manche sind bereits bekannt):

1. Deformation

$$\varphi(x) = x + u(x)$$

Hierbei ist $u(x)$ die Verschiebung.

2. Verzerrung

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

Idee: Allgemein gibt, wie auch ähnlich in der Areaformel zu finden, $\frac{1}{2}(\nabla\varphi\nabla\varphi^t - I)$ ein Maß für Längenänderungen an (I = Einheitsmatrix). Zum Beispiel ist, wenn $\varphi \in O(n)$ eine orthogonale Abbildung, also insbesondere eine Isometrie ist, $\nabla\varphi\nabla\varphi^t - I = \varphi\varphi^t - I = 0$, d.h. es findet keine Längenänderung statt. Die $\frac{1}{2}$ steht bloß für spätere Normierungen dort.

Doch zurück zum Längenänderungsmaß. Wenn man dort φ aus **1.** einsetzt, folgt

$$\frac{1}{2}(\nabla\varphi\nabla\varphi^t - I) = \frac{1}{2}\{(I + \nabla\mathrm{u}^t)(I + \nabla\mathrm{u}) - I\} = \frac{1}{2}(\nabla\mathrm{u} + \nabla\mathrm{u}^t + \nabla\mathrm{u}^t\nabla\mathrm{u}) \approx \frac{1}{2}(\nabla\mathrm{u} + \nabla\mathrm{u}^t)$$

wobei $\nabla\mathrm{u}^t\nabla\mathrm{u}$ wegen der Annahme der Linearisierung bereits in seiner Ordnung irrelevant ist.

3. Spannung

$$\sigma(u) = \lambda \operatorname{div} u \cdot I + 2\mu\, \varepsilon(u) = \lambda \operatorname{tr} \varepsilon(u) \cdot I + 2\mu \cdot \varepsilon(u)$$

Hierbei verkörpert $\operatorname{tr} \varepsilon(u)$ eine Art Volumenänderung und $\varepsilon(u)$ wie gerade ausgeführt die Längenänderung.

4. Energie

Wie bereits bekannt definieren wie die Energie durch

$$E(u) = \int_{\Omega} \frac{1}{2}\sigma(u) \cdot \varepsilon(u)\,\mathrm{dx} - \int_{\Omega} f \cdot u\,\mathrm{dx} - \int_{\partial\Omega} g \cdot u\,\mathrm{dS}$$

oder in ihrer schwachen Formulierung

$$\int_{\Omega} \sigma(u) \cdot \varepsilon(\varphi)\,\mathrm{dx} = \int_{\Omega} f \cdot \varphi\,\mathrm{dx} - \int_{\partial\Omega} g \cdot \varphi\,\mathrm{dS}$$

5. Hyperelastizität

Hierbei ist die Energie gegeben durch

$$E(u) = \int_{\Omega} w(\nabla\varphi)\,\mathrm{dx} + \cdots$$

wobei w eine i.A. komplizierte Funktion ist. Ein relevantes Beispiel für w ist

$$w(\nabla\varphi) = \frac{\mu}{2}\mathrm{tr}(\nabla\varphi^t\nabla\varphi) - \mu\log(\det(\nabla\varphi)) + \frac{\lambda}{2}\{\log(\det(\nabla\varphi))\}^2$$

Hyperelastizität ist relevant, da die Linearisierung in manchen Aspekten keine brauchbare Näherung mehr ist. Zum Beispiel kann die linearisierte Elastizität keine Selbstüberschneidung des Materials berücksichtigen, da man bei ihr nur eine Verschiebung des Materials entlang der vertikalen (z-)Achse annimmt.

Die **Hausaufgabe** für übernächste Vorlesung (25.11.) lautet (von eCampus):

Based on the Laplacian example, write a program to solve the PDE of linearized elasticity on a square, with fixed bottom and a boundary force on top. Visualize the deformed configuration, colored by the stored energy density (i.e. the integrand of the stored energy).

You might need need the following functions / classes:

VectorFunctionSpace(…)
Constant ((0,0)) -- for vector valued constant functions
grad(u).T -- for the transposed of the gradient
Identity(2) -- for the identity matrix
inner(,) -- for the : product of matrices
project() -- to project functions onto a finite element space

Additional exercises (optional):
- Find out how to generate meshes for more complex domains. Create and solve interesting problems.
- Extend the program to 3D.
- Modify the program to solve (nonlinear) hyperelasticity.